T A N D E M

SYSTEMS REVIEW

OCTOBER 1988

VOLUME 4, NUMBER 5

BRANDIFINO

Overview of DSM

VIEWPOINT ▪ NSS

SPI ▪ EMS

DNS Data Replication ▪ SCP and SCF

SPI and EMS Interfaces

Estimating Host Response Time

Volume 4, Number 3, October 1988

Articles Editor
Susan Wayne Thompson

Managing Editor
Ellen Marielle-Tréhoüart

Associate Editor
Suzanne Ambiel

Assistant Editors
Sarah Rood
Jodi Steiner

Editorial Assistant
Natalie Lingo

Technical Advisors
Mark Anderton
Bart Grantham

Cover Art
Judith Hill
Stephen Stavast

Production and Layout
Judith Hill

Typesetting
Tandem Typography

The *Tandem Systems Review* is published by Tandem Computers Incorporated.

Purpose: The *Tandem Systems Review* publishes technical information about Tandem software releases and products. Its purpose is to help programmer-analysts who use our computer systems to plan for, install, use, and tune Tandem products.

Subscription additions and changes: Subscriptions are free. To add names or make corrections to the distribution database, requests within the U.S. should be sent to Tandem Computers Incorporated, *Tandem Systems Review,* 18922 Forge Drive, LOC 216-05, Cupertino, CA 95014. *Requests outside the U.S. should be sent to the local Tandem sales office.*

Comments: The editors welcome suggestions for content and format. Please send them to the *Tandem Systems Review*, 18922 Forge Drive, LOC 216-05, Cupertino, CA 95014.

Tandem Computers Incorporated makes no representation or warranty that the information contained in this publication is applicable to systems configured differently than those systems on which the information has been developed and tested. It also assumes no responsibility for errors or omissions that may occur in this publication.

Copyright © 1988 by Tandem Computers Incorporated. All rights reserved.

No part of this document may be reproduced in any form, including photocopying or translation to another language, without the prior written consent of Tandem Computers Incorporated.

The following are trademarks or service marks of Tandem Computers Incorporated: DNS, ENFORM, ENSCRIBE, EXPAND, FOX, GUARDIAN, GUARDIAN 90, MEASURE, MULTILAN, NonStop, NSS, PATHWAY, PS MAIL, TACL, Tandem, the Tandem logo, TXP, VIEWPOINT, VIEWSYS.

2 **Overview of DSM**
Pete Homan, Bernice Malizia, Edith Reisner

12 **VIEWPOINT Operations Console Facility**
Roger Hansen, Greg Stewart

26 **Network Statistics System**
Mike Miller

36 **Tandem's Subsystem Programmatic Interface**
Gary Tom

54 **Event Management Service Design and Implementation**
Hank Jordan, Randy McRee, Rudy Schuet

68 **Data Replication in Tandem's Distributed Name Service**
Tom Eastep

76 **SCP and SCF: A General Purpose Implementation of the Subsystem Programmatic Interface**
Tom Lawson

82 **Using the Subsystem Programmatic Interface and Event Management Services**
Keith Stobie

90 **Estimating Host Response Time in a Tandem System**
Helaine Horwitz

Overview of DSM

Distributed Systems Management (DSM) is Tandem's approach to network and systems management. It contains an architecture and a set of products. The products are based on an architecture that provides operators with an integrated view of all the resources that a business application requires. The architecture and products allow customers to automate operations functions; they allow centralized management of EXPAND™ networks of distributed systems as well as heterogeneous networks containing Tandem systems.

C00 is the first release of DSM products in the GUARDIAN 90™ operating system. It establishes the architecture, increases the level of operations functions provided by the system, and provides tools customers can use to automate operations. This article describes the DSM architecture and provides an overview of the DSM products in C00.

The Need for DSM

Tandem has offered management tools in the past, but those tools were designed for smaller systems and networks than customers have today. In 1978, a typical Tandem system supported a network of around 100 terminals. In 1988, a Tandem system can support thousands of terminals, a network can have a hundred nodes, and complexes of systems connected by the 6700 Fiber Optic Extension (FOX™) can supply over 100 MIPS of processing power.

As systems and networks grow, managing them efficiently becomes both more important and more difficult. Larger systems typically support larger volumes of messages; they also tend to be more complex, with more variation in the kinds of messages and the kinds of relationships among the components. The increased complexity of monitoring and solving problems in a large system or network typically creates a need for more operators and operations consoles.

Not only are more operators required to run complex systems and networks, but each operator requires more knowledge to run the system. The necessary training is expensive and the turnover rate is high, increasing the customer's overhead costs. The cost of mistakes made by inexperienced or overworked operators can also be high. DSM is designed to address these problems.

DSM's Importance to Tandem

The cost and complexity of managing distributed systems has become a limiting factor in network growth. Tandem's technology allows customers to install large networks of systems capable of processing hundreds of transactions per second. As systems become larger, the number of MIPS and the price/performance of those MIPS are no longer the key issues. In Requests for Proposals (RFPs), system and network management has become a major requirement.

Some users in traditional large mainframe environments have limited or reduced data center size in order to ease the operational burden (Gartner, 1987). The quality of a vendor's system and network management offerings can be influential in determining how attractive that vendor's product will be to customers whose business depends on large computing environments.

DSM Architecture

The DSM architecture provides the basis for managing large systems and networks by improving the quality and flow of information between the system software and the operations staff. The architecture has four major goals:

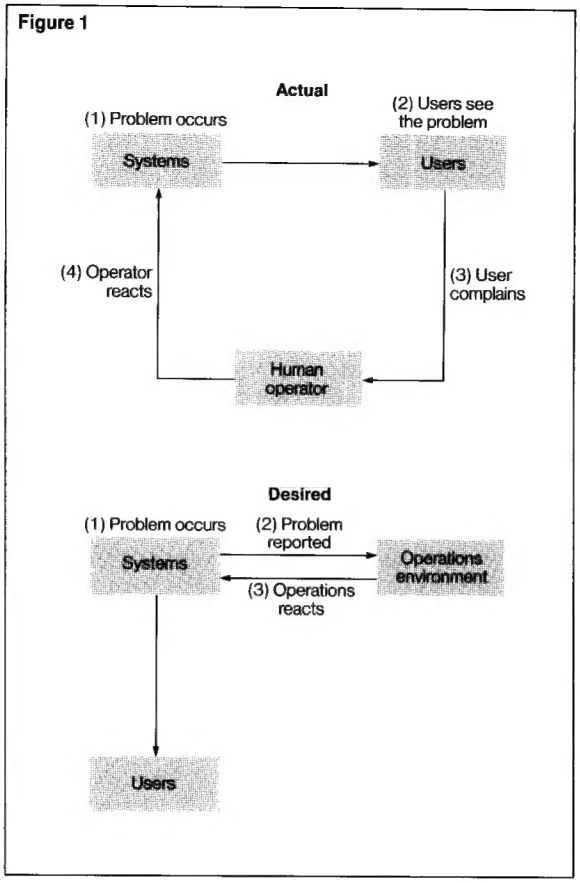
- Providing an integrated view of a system and network.
- Allowing automation of operations functions.
- Allowing flexible distribution of network control.
- Providing a migration path for customers who have developed management solutions in the past.

An Integrated View of the System and Network

Managing systems and networks involves controlling all the resources that the applications require. These resources can be local or remote; they can be hardware or software; they can be provided by Tandem, by the customer, or by a third party. This level of integration is a fundamental objective of the architecture.

Because systems are ultimately part of data networks and subject to the same management and control needs, the support structures are, and should be, congruent. (Gartner, 1987). For this reason, DSM encompasses both system management and network management. In its first release, DSM addresses the management of individual systems, of distributed systems, and of the Tandem software used to connect those systems to terminals and other devices; the DSM architecture allows for future management of other vendors' data communications equipment.

Figure 1.
Information flow in systems management.



DSM supports the management of both hardware and software functions. The components instrumented to support DSM include CPUs and peripherals, the operating system, transaction monitor, network software, and access methods. Software and hardware subsystems developed by customers can also be instrumented to support DSM.

Finally, DSM supports management of both system software and application software. Customers can use the same tools to monitor their applications and the system software on which their applications depend.

Allowing Automated Operations Functions

System managers want problems recognized, diagnosed, and fixed before the end user sees them, but this becomes more difficult as systems and networks become more complex. (See Figure 1.) Command interpreters and other operator interfaces provide a great deal of raw information; an operator must be a technical expert to understand the information and even then must follow complicated fault-isolation procedures, selecting and combining potentially relevant information from different sources.

Customers want to reduce their dependence on expert operators. They also want to manage large configurations without additional staff. Automating operations procedures in programs accomplishes both these objectives. (See Figure 2.) It also allows different companies, and even different sites, to produce operations software to meet their particular requirements.

DSM provides a set of services that customers can use to develop management applications, and defines the functions subsystems must provide to those applications. DSM defines a message format and protocol standards to ensure that subsystems offer consistent management interfaces to applications. It also includes a distributed name service (DNS™), which applications can use to insulate operators from network and subsystem boundaries and to reduce the amount of subsystem-specific information the operator must know.

Automating operations functions requires a programming investment by the customer. For cases in which automation is not feasible or not economically justified, DSM includes the VIEWPOINT™ console application, which provides a default operations environment. VIEWPOINT can coexist with other management applications or be integrated with them.

Flexible Distribution of Network Control

One of the implications of distributed processing is that although business applications and databases are split over many nodes in a network, operations expertise is available at only a few locations. It is, therefore, necessary to centralize control of multiple distributed systems, or even a whole network, at one or more locations. At that location, one or more operators share the tasks of system and network management.

DSM supports centralized and distributed network control. It allows an operator to control several nodes or a whole network from a single terminal and supports the distribution of management tasks among operators.

VIEWPOINT supports centralized management by allowing an operator to run multiple local and remote utilities simultaneously from a single terminal. On the same terminal, the operator can monitor status and events from multiple nodes. VIEWPOINT supports partitioning of operator functions by allowing different operators to receive different messages on separate terminals; each operator sees the messages relevant to the operator's responsibility.

Event Management Service (EMS) supports centralized management by allowing any node to serve as a collection point for messages that report significant events on any number of network nodes. EMS also supports message filtering so that different operators on the same node can obtain different sets of messages. Each operator—or each application that performs operator functions—can select messages that have bearing on specific problems or aspects of the network.

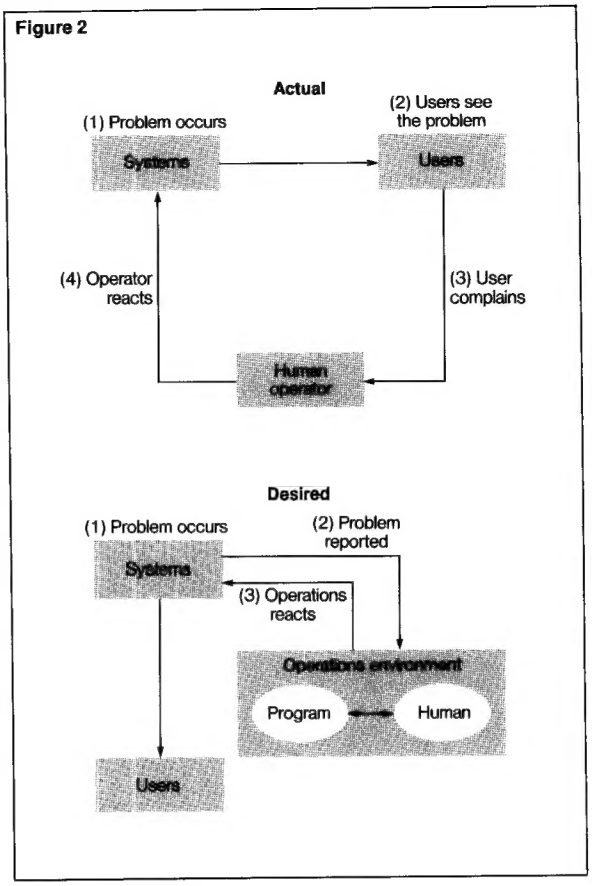
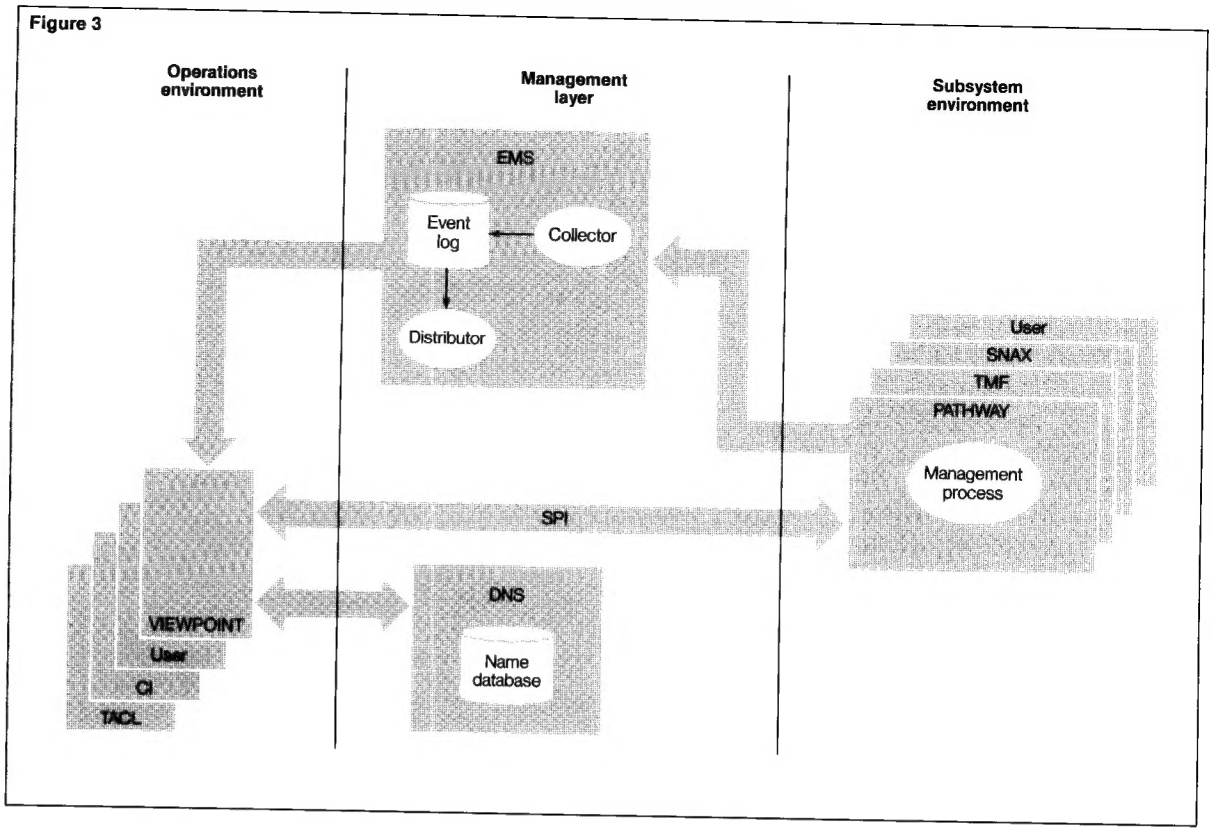


Figure 2.
Automating the operations environment.

Figure 3.
DSM environments.



DNS supports centralized management by allowing names defined on any node to be replicated on other nodes. Typically, the node where an object resides has the object name, aliases, and other information defined in its DNS database; a control node (or possibly a primary and a backup control node) has copies of the information.

DSM facilities allow autonomous control of any node; if a communications line fails between a node and the node that normally controls it, operators or applications on the severed node can control it locally.

Allowing a Migration Path

Many customers have invested considerable effort in developing custom management software using interfaces and tools that existed prior to DSM. Although DSM introduces new interfaces and even a new style of interface, older interfaces are being maintained to allow a migration path for those customers.

A Layered Architecture

One of DSM's objectives is to improve the flow of information between the subsystem environment (both system and application software) and the operations staff or the software that performs operations functions. In the DSM architecture, these concepts are formalized; the architecture consists of a subsystem environment, an operations environment, and a management layer, as shown in Figure 3.

The Subsystem Environment

The subsystem environment includes the software that directly controls devices, files, processes, and other resources used by business applications. It also includes the business applications developed by customers and custom subsystems developed by third parties. (See Figure 4.)

Subsystems—whether provided by Tandem, by the customer, or by third parties—have two ways of participating in DSM in the C00 release:

- A subsystem can provide a programmatic command interface, allowing applications to control and inquire about objects defined by the subsystem.
- A subsystem can report significant occurrences (events) in its environment to EMS, which makes the information available to interested operators and applications.

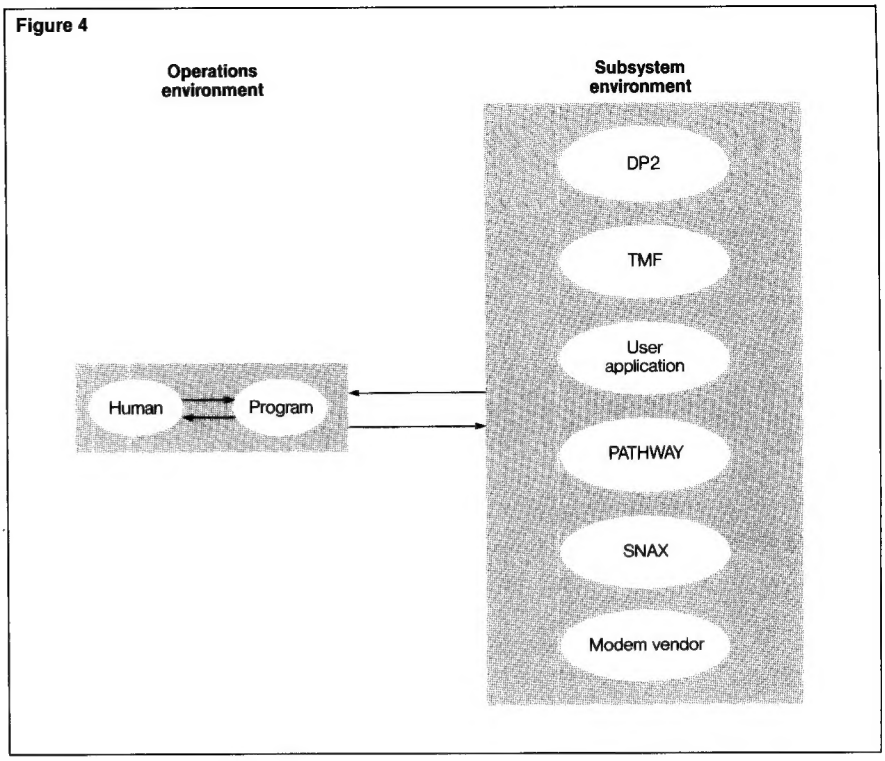
Operations Environment

The operations environment provides a management interface for operators. In some cases, it also includes applications that perform operations functions. Elements of this environment are the TACL extensible command language, the VIEWPOINT console application, subsystem command interpreters—including a new one for communications subsystems called the Subsystem Control Facility (SCF)—command files, and potentially, other management applications.

Management applications automate management functions. An application can replace one or more command interpreters, providing a simpler or more integrated operator interface. Applications can also codify procedures for accomplishing specific tasks (such as restarting terminals) or solving specific problems.

Management applications use standard application tools and reside in the same runtime environment as a standard GUARDIAN™ application. They can be written in TAL, COBOL85, or TACL, and can use the PATHWAY™ and TMF transaction management facilities and the ENSCRIBE and NonStop SQL database management products, as well as DSM services.

Figure 4



VIEWPOINT. This product is an example of a management application; it is implemented with the same set of tools available to customer applications. VIEWPOINT is generic; customers can easily tailor or augment it to suit individual needs. Key features include:

- A TACL screen from which to run command interpreters and other utilities.
- A set of TACL commands collectively called Define Process, which lets an operator maintain sessions with multiple utilities simultaneously.

Figure 4.
Tandem subsystem environment.

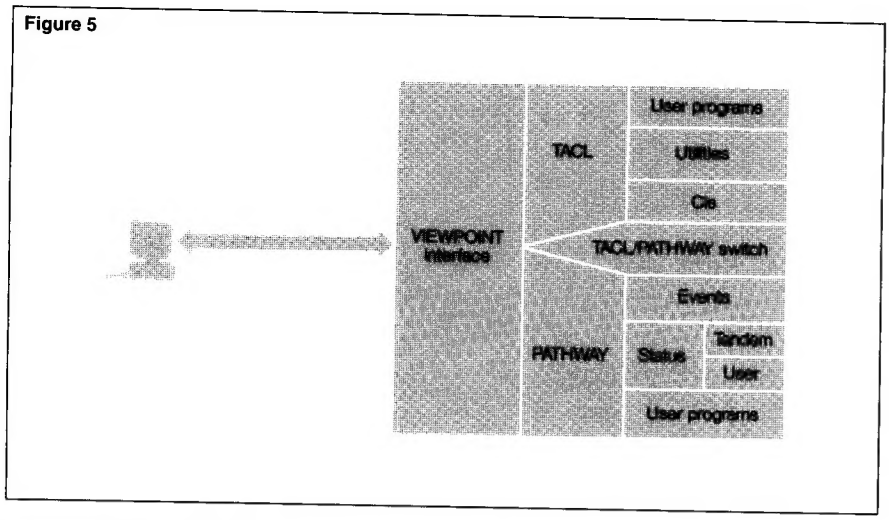


Figure 5.
VIEWPOINT.

- A block-mode screen that presents a summary of network status. The operator can specify thresholds for various status items and have the VIEWPOINT application highlight reported values that exceed the thresholds.
- Block-mode screens that display information about events on one or more nodes. The operator can monitor events as they occur or review events that have occurred since a specified time. The operator can also inquire about events involving particular objects, events demanding operator action, or events of critical importance.
- A screen that displays advisory text about a specified event. The text includes the probable cause and a recommended action.
- A clipboard facility for copying text from a block-mode screen to a text file that can be used in the TACL environment.
- A software switch that switches control of a terminal between block and conversational modes.

Because VIEWPOINT is extensible, the customer can build on it, adding extra functionality to suit particular network and systems management requirements, without having to implement a complete application. (See Figure 5.) A customer can:

- Replace the advisory text the VIEWPOINT application displays for an event, to reflect site-specific requirements.
- Write filters to determine which event messages the VIEWPOINT application will display. For example, a filter can specify which events must be considered critical.
- Add objects to the status screen and write custom server processes to obtain status information about the objects.
- Add custom commands for use on the TACL screen.
- Add custom screens, using the PATHWAY SCREEN COBOL programming language.

The Management Layer

The management layer provides the services required for management applications to control and monitor subsystem activity. It supports two kinds of interaction between a management application and a subsystem: issuing commands to control and inquire about resources in a system or a network, and monitoring events that occur in the system or network. The layer comprises the Subsystem Programmatic Interface (SPI), the Event Management Service (EMS), and the Distributed Name Service (DNS).

Subsystem Programmatic Interface. SPI is a set of procedures for building and retrieving information from command, response, and event-message buffers. An SPI message consists of a series of tokens. Applications and subsystems refer to message elements by symbolic names; they need not know the addresses or order of tokens in the message.

To send and receive SPI messages, applications and subsystems use standard file-system interprocess communication. Programmers do not need to learn about an unfamiliar transport mechanism, and they can take advantage of communication features, such as NOWAIT or timed I/O, available in the programming language (i.e., TAL, COBOL85, or TACL).

For example, the programmatic interface between an application and a subsystem is based on the requester/server model for interprocess communication. With respect to the subsystem environment, an application is a requester. The subsystem management process is the server. It accepts commands from the application and returns responses from the subsystem, providing a single management interface to subsystems composed of multiple processes.

Just as SPI does not transport messages between applications and subsystems, it does not dictate any subsystem-specific semantics. Still, by convention, subsystem interfaces that use SPI also have other characteristics in common.

Subsystem interfaces are also uniform with respect to naming of tokens and other message elements, use of subsystem-independent tokens and message constructs, process startup sequences and process-name qualifiers, and continuation of responses over multiple messages.

The message syntax defined by SPI includes a message versioning scheme to allow application and subsystem processes to communicate using different release levels of the same message definitions.

Many subsystems support interfaces based on SPI in the C00 release. New command interpreters are written as DSM management applications which use the SPI procedures and related conventions.

Event Management Service. EMS provides event-message collection, logging, and distribution facilities. Subsystems report events to EMS for logging. Applications (or operators using applications) can obtain event messages as they are logged or can review past events. The same message is available to multiple users on a node and can be forwarded automatically to remote locations.

By accepting and logging messages, EMS allows subsystems to communicate asynchronously with applications in the operations environment. A subsystem can report information to the operations environment even if no operator or application is ready to receive the information. The operator or application can find out about events as they occur or at any time thereafter.

An important component of EMS is the filter language and compiler. By using filters, applications or operators can specify the kinds of messages they wish to receive, instead of having to read or process all available messages.

Applications communicate with EMS using SPI messages. (An event message is a kind of SPI message.) Just as there are conventions that apply to all SPI messages, there are special conventions for event messages; for example, certain tokens are present in every event message, and subsystems follow published rules about when and how to report events.

Distributed Name Service. One important function a management application can provide is translating subsystem information into a form logical to the operator and vice versa. For example, an operator might regard starting an ATM (automated teller machine) as a logical action. The application must understand that because several subsystems control different aspects of the ATM, the action requires commands to multiple subsystems. The application must also be able to translate the name of the ATM (known to the operator) into the various names by which the ATM is known to the different subsystems.

To make such applications easier to write, DSM includes a name-management service called DNS. DNS manages a distributed database of names. It gives applications a standard way to store and refer to names known to subsystems, aliases known to operators, and collections of related names (e.g., the names by which the same ATM is known to different subsystems or the names of all ATMs that have the same hours of service).

An Open Architecture

One of the key features of DSM is that customers' software can be integrated into the DSM architecture. The procedures and conventions for communication between applications and subsystems are described in the DSM manuals, as are the procedures and conventions for reporting events to EMS. Thus, a user application or third-party subsystem can function as a management application or a subsystem in the DSM architecture, using the same interfaces that Tandem products use and communicating with Tandem software. (For example, an event message produced by a third-party subsystem can be displayed on a VIEWPOINT screen along with event messages from Tandem subsystems.) In a few cases, a special procedure is available to privileged software, but equivalent function is available to nonprivileged software.

Migration

DSM provides a migration path for customers who wish to convert existing applications and subsystems to use DSM tools. Compatibility features reduce the cost of conversion and the entry cost of DSM for existing customers. Four DSM features make stepwise migration feasible.

An operator using VIEWPOINT can still invoke TACL commands and macros, use existing subsystem command interpreters, and run any program that worked in previous releases. Command files and applications that send text to command interpreters will continue to work.

User and system console messages are still written to the OPRLOG, the CONSOLE, and $AOPR applications. $AOPR applications and applications that read the OPRLOG will continue to work.

User and system console messages are automatically accepted and logged by EMS, which converts them to a form suitable for filtering and distribution. The text of those messages can be displayed by VIEWPOINT and retrieved by applications.

Applications that request text versions of event messages may specify whether they want the text in the format characteristic of releases before C00 or in a new, more readable display format.

Conclusion

DSM is a significant step in the evolution of management facilities for Tandem systems and networks. It gives Tandem customers a foundation on which to integrate new functions into a single consistent approach to managing systems and networks. It includes an architecture and a set of tools on which Tandem and its customers can build.

Apart from the architecture, the focus in the C00 release has been on the functions that have the greatest impact on system availability. These are the ability to recognize problems by monitoring events in a system or a network and to resolve problems by issuing commands to subsystems.

References

Levine, P. 1987. Network and Systems Management Are Congruent. *Software Management Strategies*. The Gartner Group, Inc. T-301.373.1.

Timmins, L. 1987. SMS Conference: Automated Operations. *Software Management Strategies*. The Gartner Group, Inc. E-034-311.1

Event Management Service (EMS) Manual. Part no. 84092. Tandem Computers Incorporated.

Distributed Name Service (DNS) Manual. Part no. 84093. Tandem Computers Incorporated.

Distributed Systems Management (DSM) Programming Manual. Part no. 82587. Tandem Computers Incorporated.

Introduction to Distributed Systems Management (DSM), Part no. 84091. Tandem Computers Incorporated.

Operator Messages: Distributed Systems Management Display Format. Part no. 84103. Tandem Computers Incorporated.

Subsystem Control Facility (SCF) Reference Manual. Part no. 84159. Tandem Computers Incorporated.

VIEWPOINT Manual. Part no. 82592. Tandem Computers Incorporated.

Pete Homan has been with Tandem since 1981 and worked on the design of DSM in C00. He has over 15 years of experience in the design of transaction processing systems.

Bernice Malizia joined Tandem in 1985. She is manager of DSM Software Development and has over 15 years of experience in systems software development.

Edith Reisner has worked for Tandem since 1981. She works in the Software Publications department as a technical writer.

VIEWPOINT Operations Console Facility

The VIEWPOINT™ multifunction operations console facility is a Distributed Systems Management (DSM) application that allows operators to manage a network of Tandem systems from a single terminal. VIEWPOINT gives the operator an integrated view of the status of a group of distributed systems as well as events occurring on the systems. In addition, VIEWPOINT is easily distributed, has fault-tolerant operation, supports multiple terminals, and provides user-customization features.

VIEWPOINT is an example of an operations management application implemented using the services available with DSM. The Event Management Service (EMS) is used in the collection and presentation of events, and the Subsystem Programmatic Interface (SPI) is used in the collection of status from various subsystems. VIEWPOINT's facilities can be easily extended and may be used as a foundation for building other operations management applications.

This article provides background on the operations requirements that led to the design of VIEWPOINT and provides a brief description of the significant features. For users interested in extending VIEWPOINT or developing a similar management application, this article also provides a high-level discussion of the implementation of the status and event display environment, the command and control environment, the integration of these environments, and the provisions for extensions to VIEWPOINT.

Operations Requirements

The VIEWPOINT product answers two general sets of requirements within the DSM product family: First, users of Tandem networks required a multifunction application that would provide an integrated view for managing distributed systems. Second, given new system management facilities in DSM, an application was required that would present these capabilities and provide a foundation for further application development.

A study of customer needs and the acceptance of prototype facilities led to more specific operation requirements. These are listed below.

Centralized View
DSM and VIEWPOINT apply to a broad range of system management environments. Large-network operations control centers often require several terminals dedicated to the network management function, and in smaller environments, a single terminal is used for all system and network management tasks. In response to this, VIEWPOINT allows a single-terminal configuration as well as multiple-terminal configurations.

Each terminal running VIEWPOINT may also be used to run other network management facilities such as Network Statistics System (NSS™), VIEWSYS™, or user-developed applications. VIEWPOINT event monitoring, status monitoring, and command control screens are all available from a single terminal. Function keys may be used to move between the available screens.

Event Monitoring
Operators require that significant events occurring on a system or network be displayed in a timely and useful manner. Event displays are needed for different purposes, such as monitoring events as they occur, evaluating historical events for analysis of the sequence of events at a particular time, and looking at all events associated with a resource for problem analysis. VIEWPOINT's event screens address each of these event management needs. In addition, highlighting and outstanding event counts ease the job of ensuring that all significant events are properly handled.

Status Monitoring
Operators need to be able to monitor the availability and usage of critical resources on local and remote systems. Monitoring tools such as VIEWSYS provide detailed statistics for a selected system or CPU. In managing distributed systems, the operator needs to see high-level status indicators from several systems on one display. These indicators allow the operator to focus on a particular system or resource only when a problem is apparent. The status monitoring displays must be configurable because systems, networks, and operator responsibilities vary. The most critical resources are often user- or application-defined.

VIEWPOINT status monitoring screens provide configurable displays of high-level resource statistics. For example, the user can add statistics to the display and set thresholds for conditions that require attention.

Command and Control
Resource control capabilities currently reside in numerous subsystem and application control processes. Operators must be able to efficiently issue commands to several subsystems on local and remote systems. This must be done without requiring the operator to remember the various subsystems and resources involved in a complex operation. Operations such as START_ALL_ATMS or START_BACKUPS should perform the necessary subsystem control without requiring the operator to remember names and syntax.

These requirements imply that a powerful command facility be available to operators. TACL (Tandem Advanced Command Language) provides the base for such a facility. VIEWPOINT provides additional routines to simplify subsystem control and provides access to this and other control aids through the TACL screen.

Availability
An effective management application must be continuously available, especially when there are problems with critical resources. VIEWPOINT uses the fault-tolerant features of Tandem NonStop™ systems and the GUARDIAN 90™ operating system to ensure continuous availability.

Extensibility
The complete list of requirements for operation management application varies with each system and network installation. It is important that provisions are made for easy addition to the capabilities of the application. VIEWPOINT contains several such provisions. Fundamental in this regard is the choice of PATHWAY™ (a transaction control system) and TACL for the implementation of VIEWPOINT. Both of these products provide well-defined, high-level programming tools that may be used when extending VIEWPOINT.

VIEWPOINT Features

The preceding requirements led to the functional design of VIEWPOINT, which is briefly summarized below.

Event Displays
VIEWPOINT displays event monitoring and analysis functions on four screens:

- The Primary Event screen provides a continuous display of events occurring on the system or network.
- The Alternate Event screen shows events that are selected by the operator.
- The Last Events screen displays recent events for a particular resource.
- The Event Detail screen displays further information about any selected event, including its probable cause and the recommended course of action.

Certain events are classified as critical events. These events represent significant changes that may require attention. Events that are classified as critical are highlighted on the Primary Event and Alternate Event screens. These events may be displayed separately or they may be included in a comprehensive event display. A count is maintained of all outstanding (i.e., unacknowledged) critical events.

Status Displays
VIEWPOINT's block-mode status displays show a summary of the availability and usage of local and remote system resources. The displays are configurable in that the items sampled, the sample interval, and the titles displayed with the samples can be specified from configuration screens. Threshold values may be configured, which will cause status items to be highlighted when specified values are exceeded. A status collection server provided with VIEWPOINT gives status for CPUs, disks, lines, TMF (Transaction Monitoring Facility) transaction rates, and resource counts for PATHWAY and SNAX (SNA Communications Services). The user can also add to the set of items sampled.

Control Facilities
Using the conversational TACL screen, the VIEWPOINT user can perform control operations and run other command interpreters or management applications. VIEWPOINT provides the Define Process library, which enhances the TACL environment by allowing the operator to maintain concurrent sessions with multiple operations utilities on the local system or on remote systems. TACL and Define Process may also be used to replace multistep commands with a single command. These facilities make it possible to build custom operations procedures and policies.

Integration Features
The block-mode display screens and the conversational TACL screen are integrated to ease navigation among the screens and to facilitate information flow. A single function key moves the user from screen to screen, and commands and options can be passed when moving from screen to screen. A "clipboard" facility allows operators to copy information from the block-mode displays to the conversational environment.

Configurability
Using configuration screens, the operator can configure event and status displays. Event and status configuration files save the configuration values. Operators may establish configuration files to be associated with their user ID. Configuration values recorded in these files will automatically be established whenever the operator invokes VIEWPOINT.

Extensibility
VIEWPOINT can be extended to meet unique operations requirements. Provisions are made for the following extensions:

- Custom commands may be built using TACL and Define Process.
- Custom screens and their associated servers may be added through the EXTRAS program unit interface.
- Advisory text displayed for an event may be changed to reflect site-specific procedures.
- Default event filters may be replaced to change the set of events displayed by VIEWPOINT or to change which events are considered critical.
- Custom status collection server programs may be added to obtain status information about site-specific resources.

Display Environment

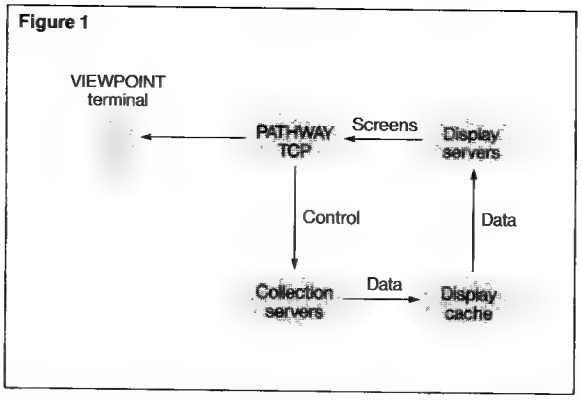
Block-mode displays are used for event and status screens and their associated configuration screens. PATHWAY provides the terminal and server management for this display environment. The following is a discussion of design decisions made in implementing this environment.

Based on PATHWAY
The requirement for a fault-tolerant, multiple-terminal application environment made PATHWAY a natural candidate for the implementation of VIEWPOINT. PATHWAY is a widely used application environment, and as a result, most users installing VIEWPOINT are able to apply existing knowledge in configuring, managing, and extending the VIEWPOINT PATHWAY application.

Two VIEWPOINT requirements distinguish it from traditional PATHWAY applications: First, users must be able to run existing applications sharing a terminal with VIEWPOINT, and second, the VIEWPOINT block-mode screens must be updated as events occur on the system or network. In the first case, a "hot key" facility was developed that allows users to navigate from VIEWPOINT's base screens to a TACL environment from which any other application can execute. The second requirement, the need to respond to asynchronous events, was answered by a new PATHWAY feature: unsolicited message processing. This feature allows PATHWAY screens to be updated as events occur.

The VIEWPOINT display environment is a standard PATHWAY application. This application may be added to a larger application environment or extended with the addition of new custom requesters or servers. A prefix, ZVPT-, is used on all VIEWPOINT server and requester names to avoid name conflicts with elements that do not belong to VIEWPOINT.

Figure 1.
Disk cache files are used to improve display response for event and status displays. Data flows from collection servers to disk cache, and then to display servers. Display servers prepare data portions of screen images for display by TCP (Terminal Control Program).



Collection and Display Servers

The principal tasks performed by this VIEWPOINT application are collecting management information (events and statistics) and displaying it in meaningful ways. The principal servers used are the collection servers and display servers. The collection servers collect events and status information, which is used to update display cache files. The display servers use these cache files to provide displays upon request. (See Figure 1.) Note that the VIEWPOINT event collection servers are distinct from EMS collector processes, such as $0. EMS collector processes record events that are then distributed to many clients, such as the VIEWPOINT event collection servers.

Two event collection servers are used: The Primary Event collection server and the Last Events collection server. These two servers execute the same program; server parameters direct the servers to perform their respective functions. The Primary Event collection server starts event distributors for the Primary Event and Alternate Event displays and updates a disk cache with event buffers received from these distributors. The Last Events collection server starts an event distributor for the Last Events display and updates the Last Events disk cache.

The status collection server class retrieves statistics required for the status display and stores the data in a status disk cache. Custom status collection servers may be added to extend the set of statistics available.

The event and status display servers read their respective cache files and produce the text for the event and status displays.

Cache Design

ENSCRIBE key-sequenced files are used for the event display cache. The organization of the event cache allows fast access to an arbitrary display page of recent events and also allows the disposition of events to be recorded with the event. The disposition states (critical, action, outstanding) are used to determine display attributes of the event.

Events for the Primary Event and Alternate Event displays are stored chronologically. The key is composed of the display type (primary, alternate, and terminal name) and an ordinal (a rotating count) indicating the order of the events.

The events in the Last Events cache are stored by subject, allowing rapid retrieval of those events that relate to a given object. The EMS subject token identifies the subjects to which an event is related. In addition to the subject, the key includes a rotating count that indicates the chronological order of the events and permits a limit to be placed on the number of events retained for a given subject.

The status display cache improves the performance of the status display. It also permits interval-based statistics to be computed, such as unit-busy percentages.

Figure 2

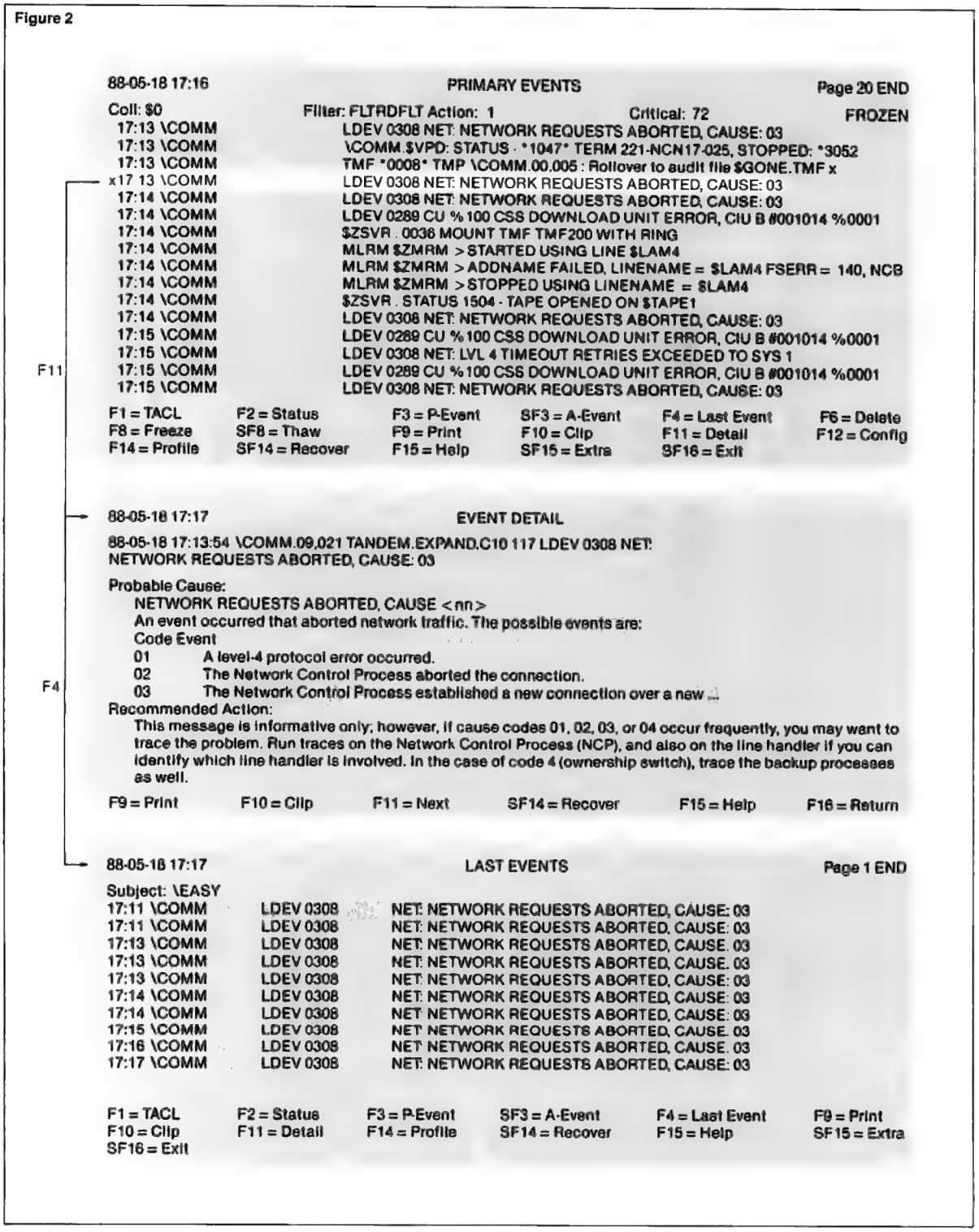
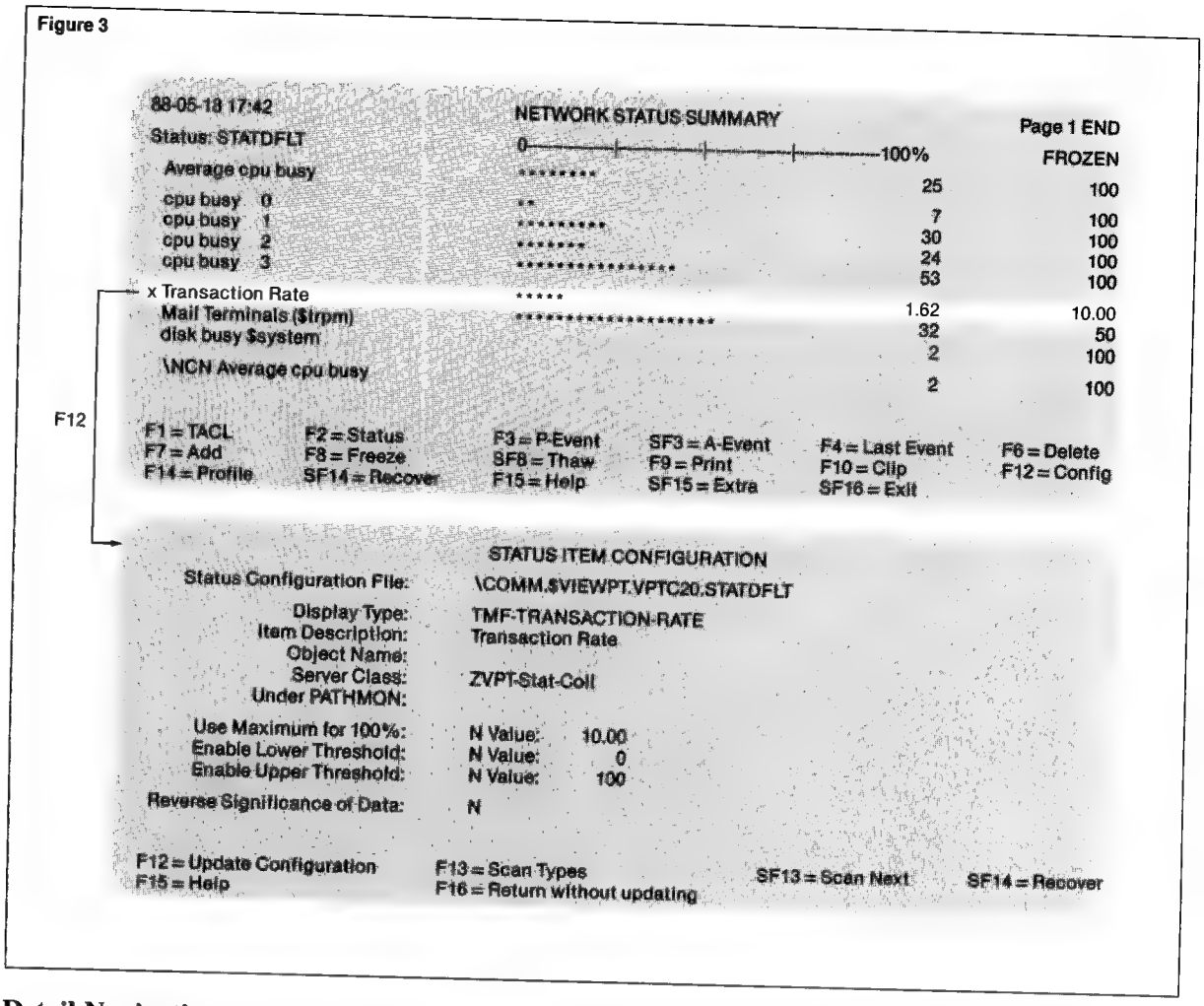


Figure 2.
Events may be selected to navigate to displays showing detail event information or to list all related events.

Figure 3.

Status items may be selected to navigate to a display showing currently configured values for the item.



Detail Navigation

The screen navigation scheme used by VIEWPOINT intentionally has a similar "look and feel" to the scheme used in Tandem's PS MAIL™ 6530 product. Event and status displays are organized as lines of data with a selection column on the left of the screen. The selection of an item on these display screens may be used to obtain a new level of detail on the item or it may be used to take action on the item. (See Figure 2.) Some of the available display function keys are listed below:

- Detail function key. Displays detailed information for an event including the full text returned by the EMSTEXT procedure and any "cause" or "recovery" text that has been provided for that event.
- Last Events function key. Displays a list of events related to a given event. The subject of the selected event is used to retrieve recent events that have the same subject. Events displayed on this screen can be selected again for detail information.
- Configure function key. Displays the status item configuration screen, which allows changes to the item description, threshold values, and other display characteristics. (See Figure 3.)

On-Line Messages

Text displayed on the Event Detail screen includes probable-cause and recommended-action text from an event messages database. This database is composed of two ENSCRIBE key-sequenced files. Each record in the files contains text indicating the probable cause and recommended action for an event. The record keys are a combination of a subsystem ID number and an event number.

One of the key-sequenced files, EVENTTD, which is delivered with VIEWPOINT, contains text that is also available from the Tandem manual, *Operator Messages: DSM Format*. The second file, EVENTCX, which is not required but may be supplied by VIEWPOINT users, may be used to override text supplied with VIEWPOINT or to provide descriptions for events generated by applications or subsystems added to the Tandem system.

Control Environment

TACL provides a control environment that is easily reached by pressing a function key from the block mode display. In addition, TACL provides several screen integration features and serves as the link to other applications and subsystems.

Programmability

TACL is integrated with VIEWPOINT so that it is accessible from any of the main VIEWPOINT screens and yet retains all of its functionality.

TACL was chosen as the VIEWPOINT command processor for two reasons: First, it offers complete text and process manipulation functions, and secondly, it provides interfaces to GUARDIAN 90 and all Tandem and user subsystems through subsystem control processes and the Subsystem Programmatic Interface (SPI). The user may create TACL macros to simplify and enhance operations while reducing operator error.

Define Process

The VIEWPOINT Define Process library enhances the TACL environment by allowing several subsystem control processes (e.g., FUP, Subsystem Control Facility, PATHCOM) to run concurrently in the background. In addition, it manages these background processes while allowing numerous user customization exits.

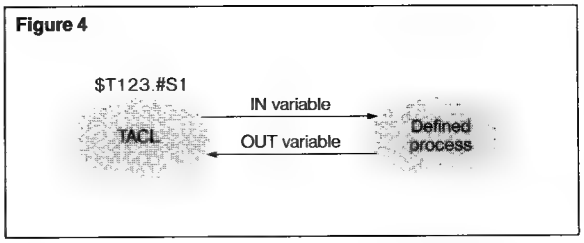


Figure 4.
Communication with a defined process. The process opens TACL using its process name and a qualifier. Commands to the process are queued in the IN variable and sent to the process as requested. Process output is queued in the OUT variable. Define Process works best with processes that treat terminals and processes identically for I/O, which encompasses most Tandem processes.

Most Tandem subsystems provide their own dedicated subsystem control process. Examples are FUP for the file system, PERUSE and SPOOLCOM for the spooler, TAPECOM for the tape subsystem, and PATHCOM for PATHWAY. This approach offers the advantage of modularity with trade-offs in command set complexity and response time.

Command set standards help to limit complexity, but the time required to start a subsystem control process (NEWPROCESS) can become a significant factor when managing distributed systems. A NEWPROCESS may take a few seconds, or longer if a process is being started across the network, and commands often must be issued to several subsystems in order to solve a given system problem. Define Process eliminates unnecessary NEWPROCESS wait time by keeping processes running and ready to receive commands.

Define Process is based on the implicit #SERVER feature of TACL. This feature allows TACL to simulate a terminal by starting a process and setting the IN and OUT files to the TACL process name appended with a subdevice qualifier. TACL logically links the process's IN and OUT files with IN and OUT variables, places the process's last prompt in a PROMPT variable, and inserts a keyword for the process's status in a STATUS variable when the process stops. (See Figure 4.)

Figure 5

```
23>dp peruse /pname p/
pstart: starting $system.system.peruse process
24>p j

JOB  BATCH  STATE   PAGES  COPIES  PRI  HOLD  LOCATION  REPORT
931         READY   192    1       4          #HOLD     CONTENT
953         READY   192    1       4          #HOLD     ALL
25>  undp p
```

Figure 5.
Defining, using, and undefining a process. PERUSE is defined and started, a list of jobs is printed, and the process is then undefined.

Define Process monitors these I/O variables and prompts the user on behalf of the process. This makes it appear to the user as though the process is directly prompting the terminal.

The IN and OUT variables act as queues. Commands to a defined process are queued in the IN variable as separate lines. When the process prompts TACL for the next command, TACL will extract the first line of the IN variable and pass it to the process. By queuing commands for a process, a user may issue NOWAIT-style commands to a process. Define Process uses the PROMPT and STATUS variables to prompt the user and to discover when the process stops so that it may be restarted.

Each command is usually passed through to the process but may be manipulated using several user exits. Define Process can trap custom-abbreviated commands and expand them to full subsystem-specific commands. It can automatically react to process output, recall previous commands using TACL's HISTORY buffer, make use of function keys, and redirect input or output to a file or variable.

Providing "persistent" availability, Define Process will restart a process if it stops for any reason; the user need not be concerned if a defined process is running. Commands may be sent to several subsystem control processes simultaneously to take advantage of parallelism. Output from a process may be parsed by the user using TACL's string manipulation commands.

Command Building Primitives. Define Process provides TACL routines to manipulate defined processes. The routines, which can be manually invoked at a TACL prompt or automatically invoked through a custom routine, allow the user to:

- Define and "undefine" processes.
- Start and stop processes.
- Redirect I/O.
- Synchronize processes after a series of NOWAIT commands have been issued.
- Check process status.
- Get help.

When a process is defined, the user gives it a name. The name can be up to 32 characters long and contain any characters that are legal in an unqualified TACL variable name. The user uses this pseudonym to refer to the process when executing a Define Process routine and to invoke the process when sending it commands. To do this, the user invokes the process name at a TACL prompt in the same manner as programs such as FUP. (See Figure 5.)

Commands to a defined process can be monitored and altered by invoking one of three types of TACL routines:

- FKEY option. If a function key is pressed, the variable associated with the function key may be invoked.
- MACRO option. When a process is invoked, a macro may be called to issue commands to the process in order to bring the process up to date, such as setting the user's current subvolume.
- PREPARSE option. Each time a command is passed to the process, a routine may be called to monitor or manipulate the command. (See Figure 6.)

Figure 6

```
29>p dl

TITAN2    DEV STATE: WAITING      FORM:
LAND2     DEV STATE: WAITING      FORM:
ITAL2     DEV STATE: WAITING      FORM:
BOOK2     DEV STATE: WAITING      FORM:
SLIDE2    DEV STATE: WAITING      FORM:
30>pinfo pc
Name            Program              Input   Output          Ready
pc              pathcom              empty   empty           no
31>
pc PATHCOM - T9153C00 - (15JUL87)
-32- = ss

SERVER                  #RUNNING      ERROR       INFO
ENABLE-SERVER              1
ZVPT-EVLE-COLL             1
ZVPT-EVNT-CNFG             1
ZVPT-EVNT-COLL             1
ZVPT-EVNT-DETL             1
ZVPT-EVNT-DISP             1
ZVPT-EVNT-NTFY             1
ZVPT-GARB-COLL             1
ZVPT-HELP-SRVR             1
ZVPT-STAT-CNFG             1
ZVPT-STAT-COLL             1
ZVPT-STAT-DISP             1
ZVPT-TACL-SWCH             1
-33- = fsts enable-server
SERVER ENABLE-SERVER, FROZEN
SERVER ENABLE-SERVER, STOPPED
SERVER ENABLE-SERVER, THAWED
SERVER ENABLE-SERVER, STARTED
-34- = !f
-34- = fsts enable-server
SERVER ENABLE-SERVER, FROZEN
SERVER ENABLE-SERVER, STOPPED
SERVER ENABLE-SERVER, THAWED
SERVER ENABLE-SERVER, STARTED
-35- =
35>p
-36-_time [F2]
January 15, 1988 10:45:22
-37-_
```

Figure 6.

Shortcut commands to a defined process using PREPARSE and FKEY. The first example is PERUSE (P) with a DEV LASER (DL) command. This issues a DEV command for each laser printer location and formats the output. Next, PATHCOM (PC) is started on demand, a STATUS SERVER command is issued with SS, and a FREEZE/STOP/THAW/START SERVER ENABLE-SERVER sequence is initiated with FSTS ENABLE-SERVER; TACL history is used to recall the command. Finally, a function key is used within PERUSE.

Network Control Examples

Define Process offers background process features useful for all classes of Tandem users, but it is especially advantageous for managing distributed systems. TACL and subsystem control processes such as PUP, CMI, SCF, and PATHCOM can be kept running and logged on to remote systems allowing the user to issue commands across the network without startup delay. (See Figure 7.)

Many programs offer commands to set up an environment to simplify command entry. A user-written Define Process INIT macro can automatically initialize this environment for a defined process each time it is started. Shortcuts for commonly issued command sequences can be created using the PREPARSE option and a user-written TACL routine.

The VIEWPOINT Macro Library

TACL macros are especially beneficial when they are written to meet the specific needs of a user. It is also useful, however, to build macros that meet the general needs of distributed systems managers. Generally, applicable DSM-oriented macros are included in the VIEWPOINT macro library. This library is shipped with every release of VIEWPOINT and is automatically loaded and available to all users. The library's directory (:UTILS:VPTLIB) may be placed on the TACL use list, thus making the variables available without name qualification.

Figure 7

```
44 > outvar dpstatus
pcom status pathmon
cm status cmp
dnsc status
46 > dpstatus
PATHMON — STATE = RUNNING   CPUS 0:1
    PATHCTL      (OPEN)      $VIEWPT.VPTMAIN.PATHCTL
    LOG1 SE      (OPEN)      $0
    LOG2         (CLOSED)

    REQNUM       FILE        PID              PAID      WAIT
    1            PATHCOM     \COMM.07,069     165,28
    2            TCP         $VTCP
    3            PATHCOM     \COMM.07,118     165,28
Object: CMP \COMM.$Y119
CMP STATUS
    STARTED
SWITCHES %160000
    INITIALIZED
    OPENS ALLOWED
    AUTOSTOP ENABLED
    NONSTOP DISABLED
    BACKUP NOT NEEDED
    CHECKPOINTING INACTIVE
    NETWORK MONITORING DISABLED
DNS Status at 10:47:00 , 15 Jan 1988          SYSTEM \COMM

    State: Stopped
```

Figure 7.
Commands to multiple defined processes can be combined into one command. DPSTATUS queries defined processes PATHCOM (PCOM), CMI (CM), and DNSCOM (DNSC) for general subsystem status. These processes can be running locally or remotely.

Figure 8

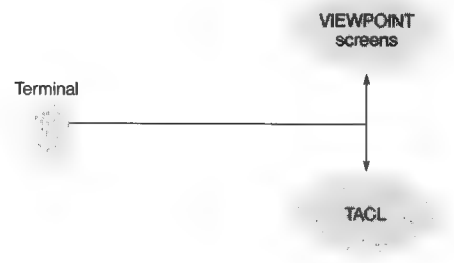


Figure 8.
Sharing VIEWPOINT block mode and TACL conversational mode on one screen.

VPTLIB contains the macro DNSINFO, which is an example of one way a user may integrate the VIEWPOINT clipboard, Define Process, and DNSCOM. DNSINFO provides a simple scheme that maps Tandem file names to DNS names. To use DNSINFO, the user simply clips events from an event screen, types DNSINFO on the option line, and presses the TACL key (F1). Then, using Define Process, DNSINFO defines a DNSCOM process and reads the events in the clipboard looking for device or file names. For each name DNSINFO finds, it will issue an INFO device command to DNS and show the DNSCOM output.

PATHWAY and TACL Integration

The DSM operator requires access to VIEWPOINT screens to monitor distributed systems as well as TACL to issue commands to subsystems; VIEWPOINT provides concurrent sessions based on PATHWAY and TACL on one terminal. The PATHWAY and TACL environments use block and conversational terminal modes, respectively, and are therefore normally mutually exclusive. This section presents design and implementation details about the integration of the VIEWPOINT environment based on PATHWAY and TACL. (See Figure 8.)

Navigation

When the user calls up VIEWPOINT on the terminal, the user's TACL process becomes linked with a VIEWPOINT session based on PATHWAY, and VIEWPOINT function key variables are set up to enable the user to navigate between VIEWPOINT screens. The user's original TACL process becomes the TACL screen until the user exits VIEWPOINT.

To move between TACL and VIEWPOINT screens, the user simply presses F1 for TACL and the appropriate function key for the desired block-mode screen. The user may pass arguments to the invoked screen in both environments in a consistent fashion.

From TACL, the user types the arguments on the TACL command line and presses the function key for the desired screen. For example, an event subject (e.g., $SPLS) may be passed to the Last Events screen by typing the subject and pressing F4. From a block-mode screen, the user may type a TACL command (e.g., PUP UP $LH) on the option line and press the TACL key to initiate the command.

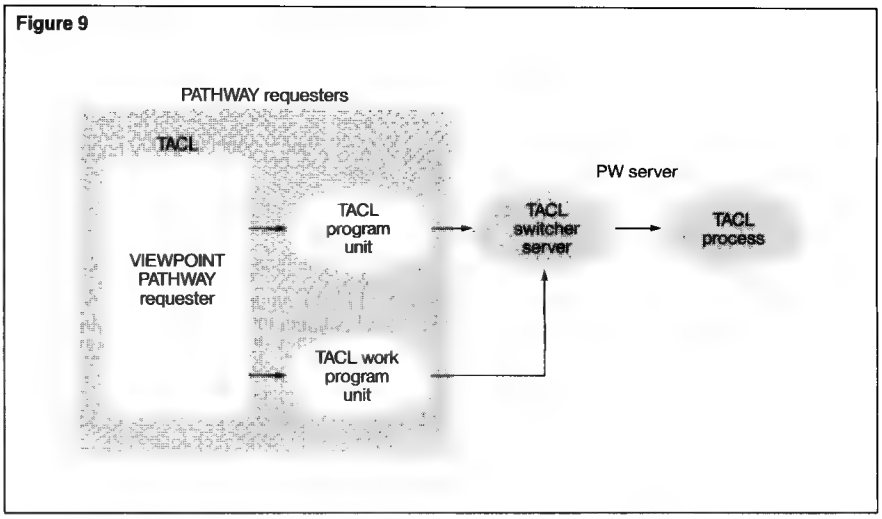
TACL Switcher Server

The key element of the switching mechanism between TACL and PATHWAY is the TACL switcher server. Supporting this server are two requesters and a TACL macro library located in :UTILS:VIEWPT. This server allows VIEWPOINT to quickly switch the terminal between the two environments and to benefit from the services of the user's logged-on TACL process.

Server Design. A VIEWPOINT (block-mode) requester may either use TACL as a server or allow TACL to take over the terminal. The requester calls one of two program units designated for each purpose and passes to the TACL switcher server all context necessary to wake up the user's TACL process. The TACL switcher server is a context-free server class; dynamic copies of the server are automatically created and managed by PATHWAY as users need to access TACL. The server is only needed while the user is on the TACL screen or TACL is performing a background service such as accessing a file. (See Figure 9.)

Inheriting User ID and Environment. The link between PATHWAY and TACL provides unique advantages to VIEWPOINT. Inherent to the link protocol is an exchange of information not usually available to traditional PATHWAY applications. Included in this information is the user's current user ID and subvolume, which allows VIEWPOINT to identify the user without requiring a logon procedure. By employing the user's TACL as a server, VIEWPOINT performs file and spooler access required for printing and clipboard operations.

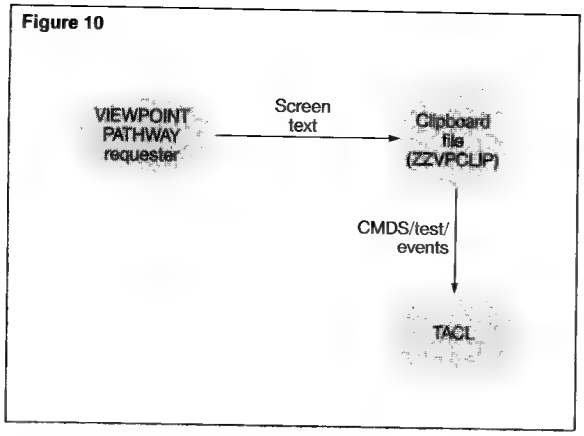
Figure 9



Data Interchange Using the Clipboard. Because the screen clears when a terminal switches between block and conversation mode, VIEWPOINT features the clipboard to assist in preserving context between the environments. An EDIT file called ZZVPCLIP on the user's saved default subvolume accumulates text taken from VIEWPOINT screens.

Figure 9.
Switch from a PATHWAY requester to the TACL screen. The TACL program unit causes TACL to take over the terminal, and the TACL work program unit uses TACL as a background server. TACL switcher communicates with TACL though a #SERVER file. TACL returns a user profile as the reply for each request.

Figure 10.
Data interchange using the Clipboard. Event Detail text may contain instructions for fixing the problem associated with an event and may contain TACL commands, which can be invoked directly.



The clipboard may be used to copy data from most block-mode screens for use in the TACL screen. Lines are selected by positioning the cursor on the desired line or marking several lines and pressing the "clip" key. Probable-cause and recommended-action instructions available on the Event Detail screen may contain TACL commands or text to assist in problem resolution. The user may preserve selected events or status items. (See Figure 10.)

The contents of the clipboard file may be displayed on the screen or processed by a utility, and the file may be purged when the information is no longer needed. DNSINFO makes use of the clipboard to supply system-to-user name mapping services. Preserving context eliminates the need to retype information, reducing errors and increasing operator productivity.

Print Operations. VIEWPOINT supplies a print command to print the screen to a file, process, or device. When the user presses the print key, the PATHWAY requester sends the contents of the screen to TACL, and the requester tells TACL where to send the screen. TACL then directs the screen to the desired location. Because TACL is performing the I/O under the user's ID, the I/O is subject to standard security checks on the user's ID.

Status Line Notification

A VIEWPOINT user may leave the PATHWAY environment for extended periods of time to run other programs such as VIEWSYS. To keep the user informed of important events, VIEWPOINT uses the 25th line of the terminal to notify the user of a critical or new action event. The text of the event is shown, along with the new total number of outstanding critical and action events. The user may heed or ignore the event without disturbing the current environment.

Extensibility

The extensibility features of VIEWPOINT facilitate the addition of new subsystems and the addition of new management interfaces. Conversational interfaces may be added using TACL and Define Process. Block-mode interfaces may be added using the EXTRAS screen.

As new subsystems are added, requiring new operator advisories, additions may be made to the event detail database. As new resources to be monitored are identified, status servers may be added.

Each release of VIEWPOINT includes a set of files containing software interfaces required to develop extensions to VIEWPOINT. Files are provided in DDL, COBOL, and TAL (Transaction Application Language) formats.

TACL and Define Process

Any TACL variable can be invoked from the TACL screen or by using the option line when invoking the TACL screen. Define Process allows complete customization of the method of starting, initializing, and interacting with a process. The combination of TACL and Define Process supplies powerful command automation facilities.

EXTRAS Screen

The EXTRAS screen may be added to VIEWPOINT by writing and compiling a SCREEN COBOL program unit named ZVPT-EXTRAS. A feature similar to one in PS MAIL 6530, the EXTRAS program unit can be a simple screen or a menu leading to a variety of added services.

Using the option line, the EXTRAS program unit interface allows commands and options to be passed to other VIEWPOINT screens. For example, the EXTRAS screen could pass a command to the TACL screen or pass a subject for event display to the Last Events screen.

Event Detail

Records may be added to the Event Detail database to override descriptions provided by VIEWPOINT or to describe events that are created by added subsystems. Each release of VIEWPOINT will contain updates to portions of the database provided by VIEWPOINT.

The *DSM Programming Manual* contains ENABLE commands that may be used to generate an application to update the database. Such an application can be a convenient addition to the set of applications available from the EXTRAS screen.

Status Server

The status collection server is a standard PATHWAY server that performs resource sampling tasks. By adding new status collection servers to VIEWPOINT, the types of resources monitored by VIEWPOINT can be extended.

In VIEWPOINT's status server, SPI encoded requests retrieve resource count values from TMF, PATHWAY, and SNAX subsystems. Using the MEASREADACTIVE interface of MEASURE™ (a performance statistic gathering facility), VIEWPOINT receives the busy values for CPUs, disks, and lines. Several other counter values such as file busy and process busy are available from MEASURE. These counters could be retrieved by collection servers added to VIEWPOINT.

Conclusion

VIEWPOINT employs the features of the DSM architecture and TACL to provide a cohesive, fault-tolerant user interface for managing distributed systems. It consolidates status and event information into a single view of all network systems and component applications. Facilities provided with VIEWPOINT allow concurrent access to multiple management utilities throughout the network. VIEWPOINT can be easily tailored to fit specific operational and organizational needs, and it provides a base for future DSM applications.

References

Distributed Systems Management (DSM) Programming Manual. Part no. 82587. Tandem Computers Incorporated.

Roger Hansen joined Tandem in 1981 to develop INSPECT. In 1985 he joined the Distributed Systems Management Group to develop VIEWPOINT. Prior to joining Tandem, he developed software for several major corporations. He holds an M.S. in Computer Science from the University of Southern California and a B.S. from Iowa State University.

Greg Stewart joined Tandem in 1984 as a software designer for PATHWAY. In 1985 he joined the Distributed Systems Management Group to develop VIEWPOINT. He holds a B.S. in Computer Science and Mathematics from the University of Oregon.

Network Statistics System

Tandem performance monitoring tools were originally restricted to statistics collection at a single node or system. Until recently, there was no mechanism to gather statistics and present a network-wide perspective of system performance. Information could be gathered on a node-by-node basis only.

Network Statistics System (NSS™), a Distributed Systems Management (DSM) subsystem, provides a "global perspective" of an entire multi-node network. NSS gives systems analysts, managers, and operators current and historic high-level system performance information, simplifying the management of two key distributed system resources: processors and EXPAND™ line handlers.

This article discusses why NSS was built, who the intended user is, and what type of information it provides. NSS configuration, threshold values, performance event generation, and database maintenance options are also explained.

Overview

NSS is a DSM subsystem that gathers information useful for managing distributed networks. Statistics on CPUs and EXPAND line handlers are collected and stored in a database. The NSS database is maintained automatically, purging and archiving data as specified by the user. Through the use of graphic displays, interactive reports, and event console messages, NSS communicates the statistics to the user.

NSS collects this information with minimum impact on system and network resources. The statistics are available regardless of network propagation delays.

NSS can be configured for a variety of network sizes. Although networks with three or more nodes benefit the most from using NSS, even a single-node "network" benefits from performance threshold events and graphic CPU performance history.

Other NSS features include:

- Current and historic reports.
- User-defined performance thresholds.
- VIEWPOINT™ threshold reporting.
- Real-time statistics database.
- Automated database maintenance.
- Distributed or central configuration.
- Text and block-mode interfaces.
- Scoreboard and graphic displays.
- Multi-node time-of-day synchronization.

Advantages

A subtle, but significant, distinction between NSS and other on-line system performance display tools is that NSS does not require a user to constantly watch a terminal to monitor system resource utilization. Instead, NSS:

- Generates performance threshold events for display on the VIEWPOINT primary events screen. The "last events" capability of VIEWPOINT permits a history of NSS events to be readily displayed. Thus, performance events are reported along with other system events on a common system console.
- Automatically maintains and displays an ongoing history of system and line-handler performance in an on-line statistics database.
- Collects statistics at a sample interval in minutes, rather than seconds. Sampling on an interval of minutes reduces the need for visual integration of statistics over time as well as reduces data collection overhead.

History

NSS was developed primarily to assist in understanding network-wide system resource utilization in a large (50-node) distributed environment. To successfully manage a network of this size from a central location, global information on processor performance and EXPAND traffic was needed.

Monitoring a Large Network

Before NSS was developed, a network of this size was monitored by periodically running XRAY™ or VIEWSYS™ measurement tools on several nodes at a time. This required interaction with as many programs as there were nodes. This approach worked well for small networks, but in larger networks it is cumbersome. Analyzing output from three programs may be manageable, but 50 is not. Automated system level reporting still required that the operator analyze as many reports as there were nodes.

Another reporting alternative, consolidated batch performance reporting, does not meet the need for current and historical on-line system resource utilization data. Batch reporting is acceptable for long-term capacity planning, but for short-term system management planning—a few minutes to a few days—an on-line summary level database is required.

In addition, in a network of this size, processor loading in each system can be variable and unpredictable. It is not adequate to check the performance of a node on a periodic basis; constant monitoring is required.

With these limitations, monitoring a large distributed network using VIEWSYS or XRAY proved inefficient and inadequate, prompting the development of NSS.

GUARDIAN Improvements

A second reason for developing NSS benefited both customers and Tandem. The internal GUARDIAN™ resource requirements of a large EXPAND network needed validation. NSS helped identify excessive utilization of GUARDIAN link control blocks and EXPAND local pool pages under certain conditions. This information led to significant improvements in the GUARDIAN operating system.

Figure 1

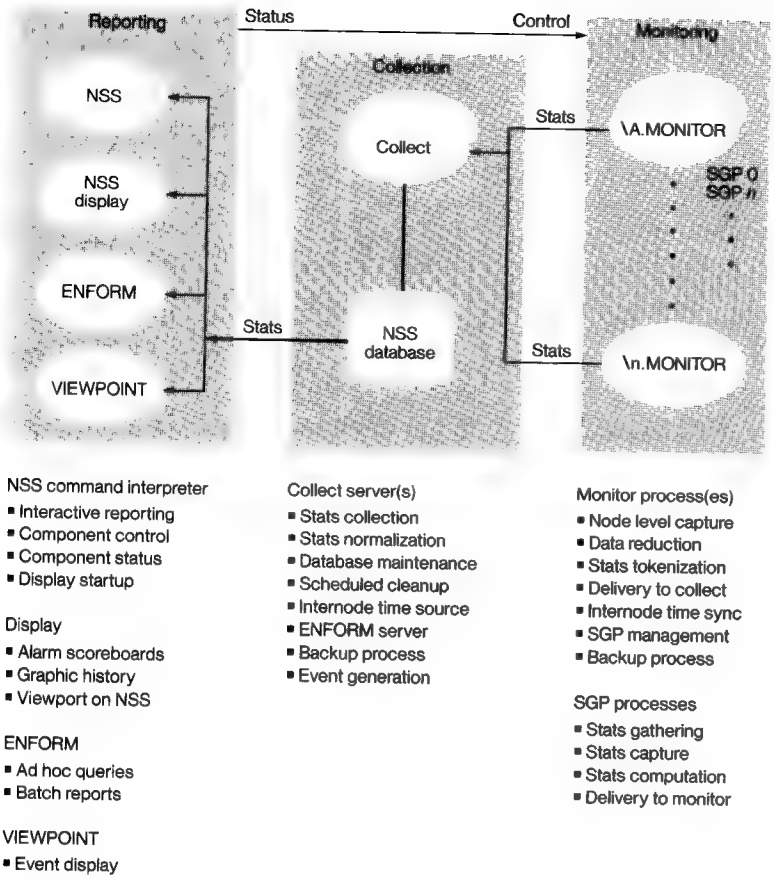


Figure 1.
NSS process diagram.

Network Management

The information provided by NSS can be used to determine where potential performance problems exist. NSS automatically provides performance and error rate information about processors and EXPAND line handlers. With this type of information, systems personnel can make more informed decisions about network management. Analyzing the information collected by NSS provides an indication of the overall "health" of a system. This information can direct system personnel to other diagnostic or monitoring subsystems such as MEASURE™ in order to gather more detailed information.

For example, NSS detects situations such as excessive CPU queueing or processor utilization in a network and reports this information to VIEWPOINT. Given this information, system operations can assign someone to investigate the situation. Or if an EXPAND line between two nodes had an increasing number of intermittent BCC errors over the last few hours or days, a message could be sent to VIEWPOINT to instruct a technician to check the line.

Using the NSS database, system management personnel can extract historical data on a variety of system performance information. Trend identification and analysis, historical usage statistics, and growth rates can be determined from the database. This information can provide valuable assistance to a capacity planning effort.

Statistics Collection

Statistics collection uses three different types of processes: the monitor process, the statistics gathering process (SGP), and the collect process. (See Figure 1.) Statistics are collected for two entities in a Tandem network: processors and EXPAND line handlers. Attributes reported for these entities are shown in Figures 2 and 3.

Monitor processes run on user-selected nodes of a network and manage node-level NSS functions. The Monitor starts an SGP in each CPU of its node. The SGP gathers statistics for its CPU and the EXPAND line handlers executing in its CPU and reports this data back to the Monitor.

The Monitor process bundles statistics for its node and sends them to its designated collect process, which may reside in the same or any other node of the network. The statistics for an entire node are normally contained in a single message sent to the collect process each sample interval. This results in extremely low additional network message overhead, even for very large networks.

The collect process receives statistics for specified nodes and stores them in a database. Database maintenance functions such as file creation, deletion, and release are provided by the collect process. The collect process also provides network-wide services such as time-of-day synchronization to its monitor processes.

The process structure described above has several important advantages:

- With data collection decoupled from presentation, data collection and archiving can continue without displaying the information.
- A statistics database permits data sharing and allows a variety of display techniques. Programs such as ENFORM™ or any other user-written program can access NSS statistics using simple database operations.
- Statistics reporting through a database separates data presentation from network propagation delays. This provides fast uniform response time to interactive user requests, typically less than 1 second, regardless of the time required to propagate statistics through the network.
- And finally, since data is directed toward the collection point, polling delays are minimized. If instead the collect process polled each monitor for data in an outward-directed fashion, as the number of nodes increased, polling delays would become significant. Unsolicited collection delays, on the other hand, are very small.

Figure 2

CPU attributes

SYSNAME	The Tandem network node name of the CPU statistics
P-KEY	Primary key: sysno, cpuno, and inverted timestamp of sample
TS	Timestamp of the sample in database local civil time
ET	Elapsed time of the sample interval in microseconds
CPUTYPE	Processor type 1 = TNSII, 2 = TXP, 3 = VLX, 4 = CLX

CPU statistics

BUSY	Percentage of the time that a processor is not idle
QLEN	Average number of processes on the ready list
DISPS	Average number of times per second that processes were executed
SEND-BUSY	Percent time that a processor spent executing SEND instruction
CHITS	Average number of disk cache hits per second in this CPU
DISCS	Average number of disk I/O operations per second

Memory statistics

SWAPS	Average page faults per second that occurred during last sample
MQLEN	Average number of processes waiting memory manager services
MEM-USE	Total number of physical memory pages locked
MEM-CNF	Total number of physical memory pages configured (locked + unlocked)
MAP-USE	Total number of mappool pages in use
MAP-CNF	Total number of mappool pages configured (used + free)

Control block stats

PCB-USE	Process Control Blocks (PCBs) in use in this CPU
PCB-CNF	Number of PCBs sysgenned for this CPU at the time of the sample
SYS-USE	Total number of syspool words (syspool) in use
SYS-CNF	Total number of syspool words available (used + free)
LCB-USE	Total number of system dataspace Link Control Blocks (LCBs) in use
LCB-CNF	Total number of system dataspace LCBs configured
TLE-USE	Total number of Time List Elements (TLEs) in use
TLE-CNF	Total number of Time List Elements configured (used + free)

Figure 2.
CPU attributes reported by NSS and corresponding DDL names.

Figure 3.
EXPAND LH attributes reported by NSS and corresponding DDL names.

Figure 3

Device attributes	
SYSNAME	Tandem network node name of line handler statistics
P-KEY	Primary key: sysno, line name, and inverted sample time
SUBTYPE	Device subtype, eg 0 singleline, 1 multiline, ...
PID1	Primary line handler (CPU,pin)
PID2	Backup line handler (CPU,pin)
LDEV	Logical device number of the line handler
Level 1 statistics	
BCC-ERR	Block Check Character error count
NO-FRAME-CNT	No Frame Buffer error count
LINE-QUALITY	Line quality (0-100%)
FRAME-CNT	Framing error count
ERROR-CNT	Level 1 error count
Level 2 statistics	
I-FRAMES-SENT and RCVD	Information frames sent/rcvd)
S-FRAMES-SENT and RCVD	Sequenced frames sent/rcvd)
U-FRAMES-SENT and RCVD	Un-sequenced frames sent/rcvd)
Level 4 statistics	
PACKETS-SENT and RCVD	Total packets (passthru + local) sent/rcvd
RLINK-SENT and RCVD	Total local logical messages sent/rcvd
FWD-PACKETS	Total packets (passthru only) sent
PACKETS-FWD	Total packets (passthru only) rcvd
Level 4 operations	
CONN-SEND and RCVD	Network connect operations sent/rcvd
TRACE-SEND and RCVD	Network Trace operations
NCPM-SEND and RCVD	NCP (network control process) messages
LRQ-SEND and RCVD	Link request operations
LCMP-SEND and RCVD	Link complete operations
CAN-SEND and RCVD	Cancel Link operations
ACK-SEND and RCVD	Acknowledgement operations
NAK-SEND and RCVD	Negative Acknowledgement operations
ENQ-SEND and RCVD	Enquiry of last message ack-ed
Level 4 msg buffer stats	
NET-CUR-IO-BUF	Current message buffer use in kilobytes
NET-MAX-IO-BUF	Maximum message buffer use in kilobytes
NET-CUR-OOS-BUF	Current Out-Of-Sequence buffer use in words
NET-MAX-OOS-BUF	Maximum Out-Of-Sequence buffer use in words
NET-MAX-COMBINED-BUF	Maximum OOS + message buffer use in kilobytes
NET-OOS-TIMEOUTS	Number of Out-Of-Sequence timeouts
PCHG-SEND	Path change msgs sent
PCHG-RCVD	Path change msgs rcvd
NET-POOL-SIZE	Total pool size in kilobytes (msg + oos)
NET-POOL-FAILS	Total number of pool allocation failures
NET-LT64	Number of messages less than 64 bytes
NET-LT128	Number of messages less than 128 bytes
NET-LT256	Number of messages less than 256 bytes
NET-LT512	Number of messages less than 512 bytes
NET-LT1024	Number of messages less than 1024 bytes
NET-LT2048	Number of messages less than 2048 bytes
NET-LT4096	Number of messages less than 4096 bytes
NET-GT4096	Number of messages greater than 4096 bytes

Statistics Presentation

The statistics gathered and archived by NSS can be accessed by four different methods. The method used depends largely upon the level of detail needed by the user and the type of presentation required.

VIEWPOINT Messages

The VIEWPOINT primary events screen displays events generated by NSS whenever user-specified performance thresholds are exceeded. When a threshold is exceeded, the NSS collect process generates a message and sends it to the Event Management Service (EMS). The message is then sent to the VIEWPOINT console. The VIEWPOINT "last events" function provides a summary of recent NSS performance events.

Interactive Text Reports

More detailed statistics can be displayed using NSS commands. These interactive text reports can be modified to report all or only selected nodes, CPUs, or line handlers in a network. Statistics reported can be real-time or for a specified time, date, and sample interval. Reports can be displayed at the terminal, directed to a printer, or sent to an edit file. Figure 4 (CPU AVERAGE report) and Figure 5 (BOSTON CPU reports) are examples of this type of reporting.

In Figure 4, the NSS command:

CPU *, AVG, S3

results in a text report that shows CPU statistics averaged across all available processors in each system. "S3" specifies that three NSS sample intervals of information are displayed. In this example, the intervals are 15 minutes; data shown covers the last 45 minutes. This report reveals BOSTON as the busiest system.

More detailed information on the BOSTON system is produced with the command:

+ CPU \BOSTON, S4

By specifying the BOSTON system, only CPU data for BOSTON is displayed (Figure 5). This report shows unbalanced processor utilization. CPU 0 (84% busy) is twice as busy as the next busy CPU, CPU 1 (43% busy).

Under the PgLk heading, the data shows that physical pages locked for the system are also unbalanced. CPU 1 has 50% more pages locked (1429 pages) than CPUs 0 and 2 (971 and 851, respectively). This is probably contributing to the swap rate of 1 per second reported in CPU 1 for the last 15 minutes.

A third NSS command generates another report on system control blocks (Figure 6).

+ CPU \BOSTON, PERCENT

This report reveals unbalanced PCB usage for the system (84% for CPU 1, 57% for CPU 0, and 44% for CPU 2). PCB usage affects memory pages locked; moving processes from CPU 1 may reduce swapping in CPU 1.

Further analysis of the BOSTON system can be conducted using MEASURE to determine what actions should be taken to balance the system. NSS has served its purpose by identifying potential and existing performance problems and directing further investigation.

Figure 4

+ CPU *, AVG, S3 !average cpu use for nodes over last three samples

DALLAS	Time	M, Q. Bsy_	Busy	Qlen	Disp	Disc	Chit	Swap	MemQ	PgLk
			%							
10/02	11:15	_ _	22		100	5	11			849
	11:00	_ _ _	28		135	7	15			855
	10:45	_ _	21		96	6	7			849
MEMPHIS	**Time**	**M, Q. Bsy_**	**Busy**	**Qlen**	**Disp**	**Disc**	**Chit**	**Swap**	**MemQ**	**PgLk**
			%							
10/02	11:15	_ _ _ _	54	2	124	7	12	1		1170
	11:00	_ _ _	32	1	87	4	7	1		1165
	10:45	_ _ _ _	37	1	99	5	7	1		1148
BOSTON	**Time**	**M, Q. Bsy_**	**Busy**	**Qlen**	**Disp**	**Disc**	**Chit**	**Swap**	**MemQ**	**PgLk**
			%							
10/02	11:15	_ _ _ _ _ _	56		342	6	41			1084
	11:00	_ _ _ _ _ _	55		340	6	40			1064
	10:45	_ _ _ _ _	52		317	5	42			1085
SANFRAN	**Time**	**M, Q. Bsy_**	**Busy**	**Qlen**	**Disp**	**Disc**	**Chit**	**Swap**	**MemQ**	**PgLk**
			%							
10/02	11:15	_ _	23	1	140	9	17	1		743
	11:00	_	10		64	4	5	1		734
	10:45	_	13		75	4	9	1		729

Figure 5

+ CPU \BOSTON,S4 !display Boston cpus over last hour (note cpu 0 busy)

BOSTON	Time	M, Q. Bsy_	Busy	Qlen	Disp	Disc	Chit	Swap	MemQ	PgLk
			%							
00x10/02	11:15	_ _ _ _ _ _ _	84	1	389	14	9			971
	11:00	_ _ _ _ _ _ _ _	85	1	417	14	9			969
	10:45	_ _ _ _ _ _ _	84	1	377	13	8			970
	10:30	_ _ _ _ _ _ _	84	1	392	13	9			969
01x10/02	11:15	_ _ _ _	43		320			1		1429
	11:00	_ _ _ _	44		323					1378
	10:45	_ _ _ _	40		303					1438
	10:30	_ _ _ _	41		314					1436
02x10/02	11:15	_ _ _ _	42		317	4	114			851
	11:00	_ _ _ _	36		279	5	112			846
	10:45	_ _ _	33		272	2	118			846
	10:30	_ _ _ _	35		281	3	121			844

Figure 6

+ CPU \BOSTON, PERCENT !show percent usage of system control blocks

BOSTON	\068	Et	Bsy	QL	Disp	Disc	Cht	Swp	MQ	Pgs	Pcb	Lcb	Tle	Sys	Map
			-%							-%	-%	-%	-%	-%	-%
00x10/02	11:15	15	84	1	389	14	9			24	57	27	4	10	16
01x10/02	11:15	15	43		320			1		47	84	46	2	10	26
02x10/02	11:15	15	42		317	4	114			28	44	27	3	7	14

Figure 4.
CPU AVERAGE report.

Figure 5.
BOSTON CPU reports.

Figure 6.
BOSTON system control block report.

Figure 7.

Processor Scoreboard. Highlighted screen areas indicate scores that have exceeded user-specified thresholds. For this example, thresholds were CPU BUSY 70, Q 2, I/O 20, SWAP 3.

Figure 7

NSSVIEW 16Jul87 Processor Score on 24 Cpus 6 Systems 10/01 12:03

System	CpuBusy	Cpu Q	Cpu IO	CpuSwap	System	CpuBusy	Cpu Q	Cpu IO	CpuSwap
DALLAS	0,14%	0,0	0,3	0,0	CHICAGO	0,46%	0,3	1,13	0,0
MEMPHIS	0,57%	1,2	0,13	0,0	CENTDIV	1,51%	1,1	5,9	1,2
BOSTON	0,87%	0,1	0,22	0,0	SANFRAN	2,34%	2,1	0,15	2,2

\DALLAS	\004	Et	Bsy	QL	Disp	Disc	Cht	Swp	MQ	PGS	Pcb	Lcb	Tle	Sys	Map
			--%							--%	--%	--%	--%	--%	--%
00x10/01	12:00	15	14		85	3	4			25	35	20	4	13	11
01x10/01	12:00	15	9		61	1	4			38	50	30	3	19	20
02x10/01	12:00	15	10		60	1	7			21	37	26	2	13	11
03x10/01	12:00	15	10		24					16	41	22	4	12	5
04x10/01	12:00	15	11		21					16	20	16	2	8	2
\CHICAGO	\005	Et	Bsy	QL	Disp	Disc	Cht	Swp	MQ	PGS	Pcb	Lcb	Tle	Sys	Map
			--%							--%	--%	--%	--%	--%	--%
00x10/01	12:00	15	46	3	304					37	41	40	5	17	21
01x10/01	12:00	15	40	2	296	13	23			49	67	45	5	21	37
02x10/01	12:00	15	6		56	2	4			21	45	26	4	10	20
03x10/01	12:00	15	4		18					33	19	10	1	2	3

To Freeze hit BREAK Type HELP or <command> RETURN Resume SF15 Exit SF16

Block-Mode Display

The NSS command interpreter also provides a block-mode display interface. Block-mode displays utilize T6530 or EM6530PC video attributes and graphics to present information in the form of video or color highlighted scoreboards and graphs. NSS text-mode commands and reports are permitted while in block mode. The Processor Scoreboard (Figure 7) and System Utilization Summary (Figure 8) are examples of block-mode displays.

In Figure 7, scores that exceed user-specified thresholds trigger screen highlighting. Users can see at a glance which areas are potential performance problems. In this example, the thresholds are CPU BUSY 70, Q 2, I/O 20, and SWAP 3.

Numbers (or scores) on the scoreboard indicate which CPU on each node had the highest value for CPU BUSY, queue length, disk I/O, and swaps. Low scores are meaningful in terms of understanding the overall utilization of a node. For example, DALLAS 0, 14% (under the CPUBUSY heading) means that CPU 0 was the busiest CPU on the DALLAS node at only 14%. Under the CPU Q heading, CHICAGO 0,3 indicates that CPU 0 had an average queue length of three over the last sample interval. Additional data is reported in the window below the scoreboard.

Figure 8 provides an overall network-wide graphic summary of average processor utilization for all available processors on each node in the network. In this example, the BOSTON node has been under constant load for the past 5 hours (through 10/1, 12:00). Note that once an hour (every four samples), CPU load increases on BOSTON and CENTDIV nodes.

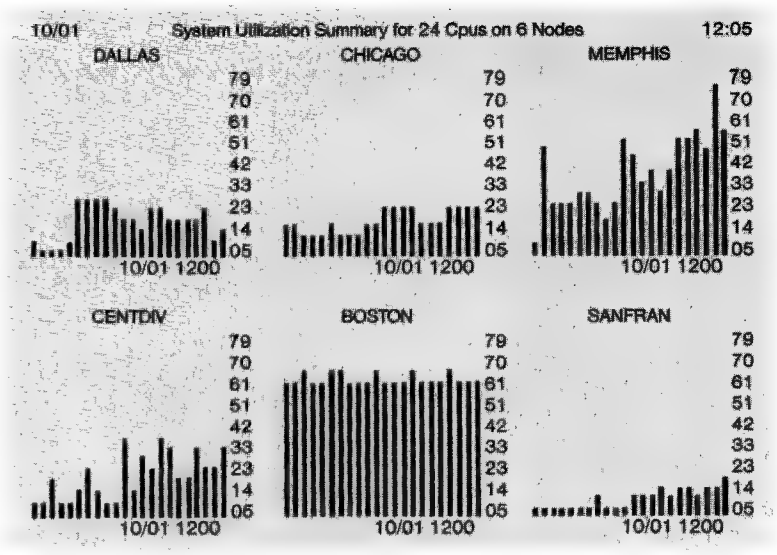
Using ENFORM

ENFORM can be used to provide customized reports based on statistics in the NSS database. This is useful for long-term historical analysis and summary reporting. NSS includes a set of ENFORM queries that users can alter to meet specific reporting requirements.

Configuring NSS

NSS can be configured in a variety of ways. Data collection can be directed to a single central node or to regional groups of nodes, or distributed so each individual node in a network has an NSS statistics database. Threshold settings and collection intervals can be modified to meet special needs. Database maintenance options are also specified by the user.

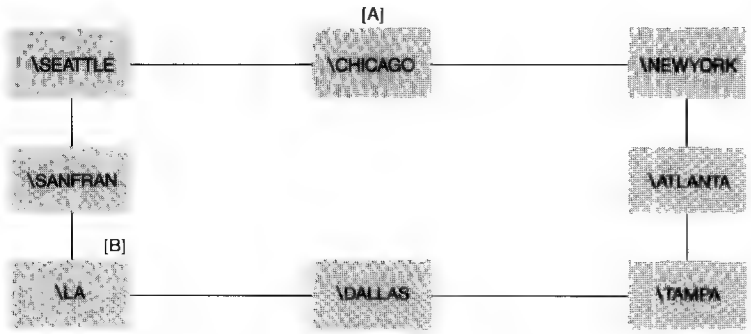
Figure 8



When executed, the NSS command interpreter automatically obeys an edit file named NSSCONF. This file contains instructions that will automatically configure NSS. The NSSCONF file can be modified using NSS commands to meet the user's needs.

Figure 8.
System Utilization Summary. This figure provides an overall network-wide system summary of average processor utilization for all available processors on each node in a network. Each vertical bar in the histogram represents one NSS sample interval (in this example, 15 minutes).

Figure 9



Example A

```
SET    COLLECT \CHICAGO      !define collection location
SET    DB $CHGO.NSS.DB       !define database location
SET    RATE 15               !sample interval 15 minutes
SET    EVENTS ON             !report performance events
SET    CPU BUSY 70, Q 2      !report Busy > = 70%, Q > = 2
SET    ID $ZNS               !ID for 1st process set

START \CHICAGO COLLECT       !start collect process
START \*                     !start monitors on all nodes
```

Example B

```
SET    COLLECT \LA           !define collection point
SET    DB $WEST.NSS.OPS      !define database fileset
SET    RATE 5                !sample interval 5 minutes
SET    EVENTS ON             !report performance events
SET    CPU SWAP 3, Q 3       !report Swap > = 3, Q > = 3
SET    ID $OPS               !ID for 2nd process set

START \LA COLLECT            !start collect process
START \LA                    !start monitors
START \SANFRAN               !on selected nodes
START \SEATTLE
START \DALLAS
```

Figure 9.
Sample NSS configurations. Examples A and B illustrate two possible NSS process configurations. Example A defines a set of NSS processes that report statistics for all nodes in the network to the CHICAGO node. Example B defines a second set of processes that report statistics for a selected regional group of nodes to the LA node.

Network Configuration

Multiple sets of NSS monitor and collect processes can be configured to operate autonomously with different designated collection locations and options. This allows geographically separate operations groups to selectively monitor different sets of network nodes. Figure 9 shows an example of such a configuration.

The ID option is used to identify a set of NSS monitor and collect processes that operate together. When only one set of processes is used, as in a single centralized configuration, only one ID is needed, and users are typically not aware of its existence. When multiple sets of processes are required, a different ID is specified for each set of processes configured. In the example shown in Figure 9, the process IDs are $ZNS and $OPS.

Collection Intervals

Statistics are sampled and collected at the interval specified by the RATE option. The sample interval typically configured ranges from 1 to 15 minutes depending on the desired frequency of reporting or statistics aggregation over time.

Statistics sampling is synchronized across all nodes by the Collect process. Sample times are organized to occur at whole minutes after the hour intervals and are kept internally in Greenwich Mean Time (GMT). For an interval of 5 minutes, sampling on each node occurs on the hour, and at 5, 10, 15, ... minutes past the hour. As long as the nodes are synchronized with respect to one another, the statistics for one node can be correlated in time to statistics from another node. NSS includes commands that provide synchronization of GMT between all or selected nodes of a network.

Performance Events and Thresholds

Performance events are messages generated by NSS when user-specified thresholds are exceeded. These threshold levels also control video or color highlighting on NSS scoreboards and graphic displays. (See Figure 7.)

Performance event threshold levels can be set for selected processor and EXPAND line handler attributes. For example, adding the following commands to the NSS configuration file causes NSS to report the busiest processor with an average CPU BUSY utilization of 70% or more over a sample interval.

```
SET CPU BUSY 70
SET EVENTS ON
```

If a processor exceeds this limit, say BOSTON.0, and it is the busiest processor on that node, then the following message would appear on the system console or the VIEWPOINT primary events screen:

```
12:00 ... INFO \BOSTON.00 87% BUSY
    for 15 min
```

Recommended CPU threshold levels for typical on-line applications are CPU BUSY 70%, QUEUE 2, SWAP 2. However, CPU threshold values are best determined by empirical observation. Different applications and end users can tolerate varying amounts of processor queuing and have different definitions of "acceptable" response time. NSS users should correlate NSS statistics with periods of observed unacceptable response time and set the NSS thresholds accordingly.

Database Maintenance Options

The NSS collect process creates a set of database files when it is first started. Because the database contains condensed statistics, one day of data for a typical node of six CPUs and four EXPAND line handlers requires only about 50,000 bytes of disk space. Thus, one day of NSS data for 20 network nodes requires just 1 Mbyte of disk storage.

The collect process performs automatic database maintenance based the RETAIN option setting.

The default retain option, RETAIN ROLLOVER, renames the current database files at the designated database CLEANTIME (typically midnight). The new name includes the month and day that the rename occurred. When RETAIN ROLLOVER is set, each set of database files represents one day of NSS information.

The RETAIN NONE option causes data for the current day to be purged at the designated CLEANTIME. This is used if only a short-term daily history is desired.

A third option, RETAIN ALL, causes all records to be retained indefinitely. Eventually the files will become full unless manual action is taken. The NSS CLEANUP command can be used to invoke manual cleanup at some selected long-term interval, such as monthly.

The fourth option, RETAIN *number*, defines the number of most recent records to retain about each CPU and line handler. With this option, a moving window of history is automatically maintained by the collect process.

Because the collect process runs continuously and assumes the ongoing responsibility of database maintenance, operation of the subsystem is automatic and does not require operator attention or scheduled maintenance.

Error Handling

The NSS INFO command is used to display a summary of recent monitor and collect process errors. This assists in trouble-shooting NSS in the event of some unusual event, such as the lack of disk space for a database file extent.

An internal NSS error log file is automatically maintained for all NSS processes that operate on a given system. These errors are also logged to EMS. Errors such as a CPU going up or down are automatically recovered. NSS will recover from all errors except for a total node failure or shutdown.

A complete list of errors by error number and parameter can be displayed with the command interpreter by entering HELP ERROR.

Conclusion

NSS provides systems analysts, managers, and operations personnel with current and historic high-level, network-wide performance views of a Tandem network.

Performance threshold event generation provides Tandem customers with information about resource demands placed upon processors and EXPAND line handlers throughout a network.

NSS is useful for identifying and locating problems, obtaining a global perspective of network performance, and directing detailed performance analysis. NSS extends these capabilities to both small and large networks, helping all Tandem customers simplify the operation and management of their networks.

Acknowledgments

The author would like to thank the members of the Tandem Memphis District for their support during the development of the initial NSS prototype.

Mike Miller is a consulting analyst for Tandem. He has performed various systems analysis functions since joining Tandem in 1978. He holds a B.S. degree in Electrical Engineering and Computer Science from the University of Illinois.

Tandem's Subsystem Programmatic Interface

Tandem's Subsystem Programmatic Interface (SPI) provides a common, message-based interface for communicating commands, responses, and event messages within Tandem's Distributed Systems Management (DSM) environment. SPI provides support for both control and monitoring functions within DSM. (See Figure 1.)

DSM application programs and DSM subsystem programs use SPI to build and interpret the command and response messages sent between them, and DSM event-message generators use SPI to build event messages describing asynchronous events. These event messages are logged and distributed by Tandem's Event Management Service (EMS), which is itself a subsystem that can be controlled through SPI. DSM event-message consumers use SPI to interpret the event messages obtained through EMS. SPI thus offers a uniform programmatic interface, not only for use by applications, but also for use within and between services and subsystems provided by Tandem and others.

Specific SPI features include:

- A common message format that supports both tokenized and structured data.
- Automated version accommodation for structured data.
- Use of Tandem's Data Definition Language (DDL) to automatically translate SPI definitions into definitions for languages such as TAL (Transaction Application Language), COBOL85, and TACL (Tandem Advanced Command Language).
- SPI procedures that can be called from TAL, COBOL85, or TACL.
- Full access to language-dependent message transport facilities such as those provided by TAL, COBOL85, and TACL.

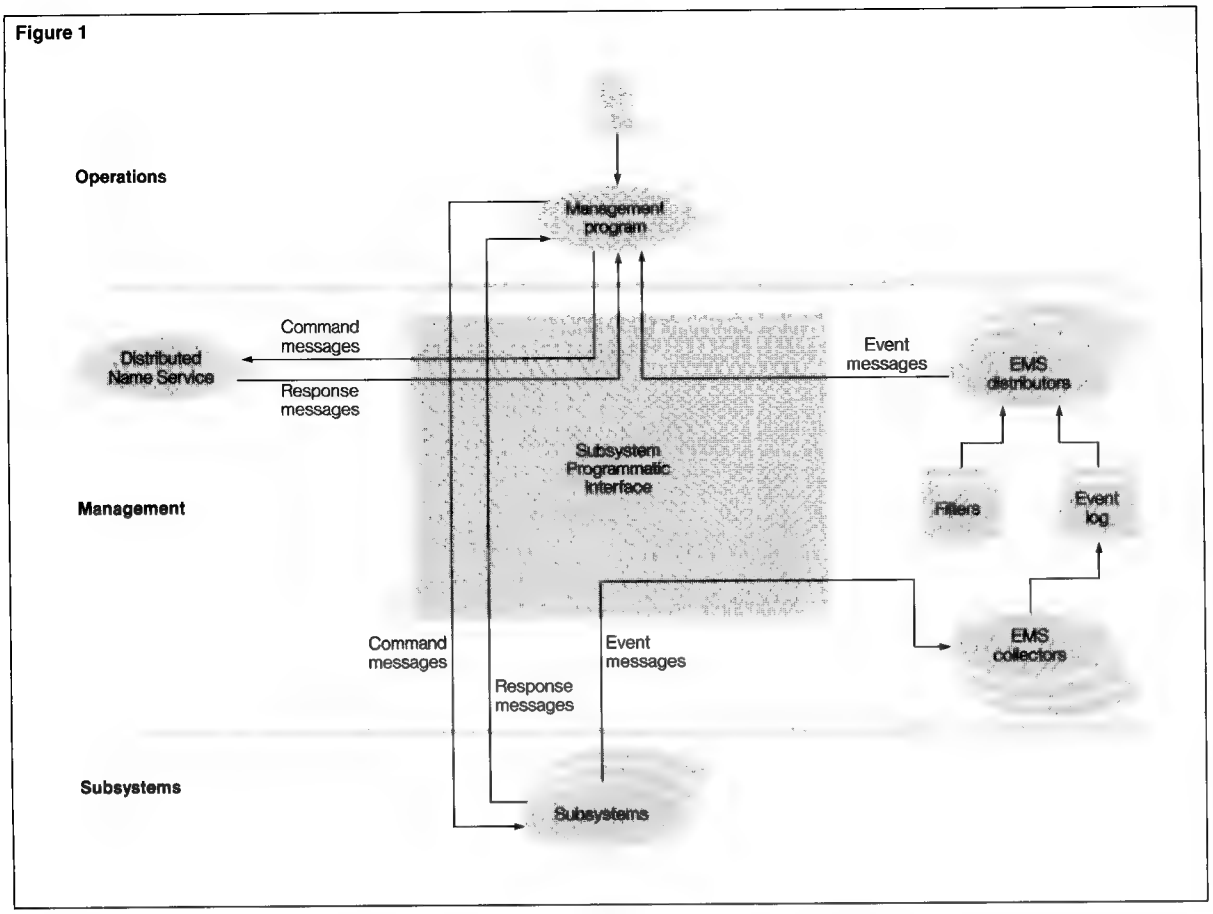


Figure 1.
The Subsystem Programmatic Interface provides a standard way to build and interpret command, response, and event messages sent between DSM components.

This article describes the interfaces used prior to DSM, discusses the design considerations behind SPI, presents an overview of SPI features, and shows how SPI meets DSM requirements. Further information about SPI may be found in the *Distributed Systems Management (DSM) Programming Manual*.

Definitions

As used in this article, *subsystem* refers to a set of programs or processes managing a cohesive set of objects. For example, in the Tandem GUARDIAN 90™ environment, FUP (File Utility Program) manages files, TMF (Transaction Monitoring Facility) manages transactions, and PATHWAY™ manages PATHWAY applications. Each of these is considered a separate subsystem.

The term *management program* is used to describe any program that helps to automate the work of a system manager or system operator. *Programmed operators* are fully automatic management programs that do not require human intervention. SPI is intended to make it easier to write management programs in general, with particular emphasis on programmed operators.

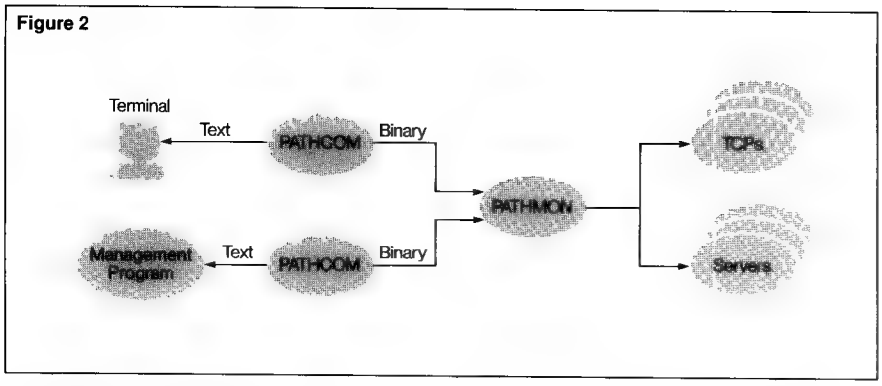


Figure 2.
Existing conversational interface (PATHWAY). Management programs using existing conversational interfaces must deal with text and attempt to simulate terminals.

Existing Interfaces

Most Tandem subsystems offer conversational text interfaces designed for use by human operators. Some subsystems, such as the SPOOLER and SORT, offer specialized programmatic interfaces as well. Each of these interfaces has some advantages, but they all present problems for management programs in a distributed environment.

Conversational text interfaces have advantages in that they are supported by almost all subsystems, they usually offer complete functionality, and they are relatively standard. They are, however, designed for use by people, not programs. Management programs must contend with the following difficulties:

- Text conversions. Internal data must be converted into command text, and response text must be converted back into internal form to be manipulated.
- Reversal of requester-server relationships. Subsystems typically expect to open a terminal or other source of conversational commands. Because a management program using this interface must simulate a terminal, it is forced into the awkward role of a server process even though it is generating the requests. This situation is shown in Figure 2.
- Need for operator intervention. Conversational interfaces assume that a human operator is available to respond to confirmation prompts and requests for operator action. This can result in prompts that are misdirected to the home terminal, prompts that are unanticipated by the management program, and commands that are rejected because they are only for interactive use.
- Version dependencies. Text displays typically change whenever the need arises. New fields are added, old fields are deleted, and existing fields are rearranged. A management program that looks for response information on a particular line or column may mysteriously fail because of a minor change in the format of a text response. This is perhaps the most serious problem because it makes programs that must handle different subsystem versions in distributed locations extremely difficult to maintain.

Existing programmatic interfaces, though designed for programmatic use, pose other problems for management programs:

- Limited or missing interfaces. Many subsystems either lack their own programmatic interface or supply an interface that cannot perform all the functions available conversationally.
- Lack of I/O control. Most existing interfaces consist of procedure calls that perform their own waited I/O. This can be a problem for management programs that must use timed or no-wait I/O to prevent deadlocks or for multithreading and performance.
- Accessible from only one language. Many of the existing interfaces are difficult or impossible to use from programming languages other than the one in which they are written.
- Nonstandard between subsystems. The biggest problem with the existing programmatic interfaces is that the interface for each subsystem is completely different from that of every other subsystem. These differences present a barrier to any program that needs to manage multiple subsystems; any time a new nonstandard interface is added, all these programs must be explicitly changed. This problem is especially acute for interpreters that want to provide the interfaces without having to understand their semantics.

Even if these difficulties could be overcome, the use of subsystem-specific interfaces would still not satisfy the DSM requirement of allowing third-party and user-written subsystems to be easily integrated with Tandem subsystems. Interpreters supplied by Tandem could not offer built-in access to customer-supplied interface procedures, and the different programming techniques required would make it extremely difficult to write generalized management programs.

SPI Requirements

DSM required an interface that would address these requirements. The most important requirement was commonality as a means of coping with the diversity of distributed systems. The interface had to be:

- Common across Tandem subsystems.
- Common across different programming languages.
- Available to user subsystems.
- Able to accommodate version differences.

To support distributed subsystems, allow for asynchronous events, and permit user control of I/O, the interface also had to be:

- Message-based.
- Independent of message transport.

Finally, for ease of use by management programs and to accommodate layered subsystems, the interface had to:

- Support management programs as requesters and subsystems as servers. This is illustrated in Figure 3.
- Use programmatic data formats (binary).
- Provide multiple contexts within a message.

Answering the above requirements, SPI includes the following features:

- The access procedures and data declarations define a common message format that can be used in TAL, COBOL85, and TACL programs.
- Users have complete control over the I/O mechanism permitting SPI use with both requester/server interactions and event logging to disk.

Figure 3

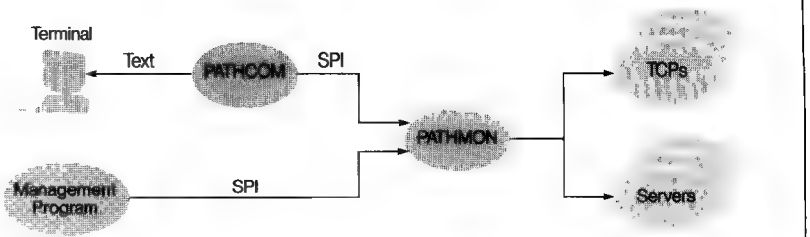


Figure 3.
Subsystem Programmatic Interface connection with PATHWAY. Management programs use a standard binary interface to interact directly with the subsystem.

- Standards provide commonality across different subsystems.
- Version accommodation provides commonality across distributed components at different release levels, as well as for old programs compiled with new declarations.

The remaining discussion will focus on how SPI components, standards, and related tools were designed to provide these and other benefits.

SPI Design Considerations

Many design alternatives were considered in the development of SPI. In addition to purely technical considerations, the practical considerations of converting existing subsystems and cooperating with existing interfaces were also very important. By testing different SPI design ideas against the needs and requirements of real subsystems as well as the architectural needs of DSM, the developers ensured that SPI would be workable in practice as well as in theory. This section of the article provides an overview of the key design decisions.

Customized vs. Common Message Format

Given that SPI was to be message-based, the first design consideration was whether the format of the buffer should be customized by each subsystem or common across all subsystems. The customized approach involved allowing each subsystem to supply its own version of the standard access procedures. The SPI procedures would then act as a big switch calling the appropriate set of procedures depending on the subsystem being used.

This approach had the advantage that existing subsystem servers would not have to be converted to use the new interface. The SPI procedures would build and decode message buffers using subsystem-defined formats and would hide the format differences from the requesters. The disadvantage of this approach is that subsystem-specific code would have to be written and bound into the SPI interface by each subsystem. Furthermore, a common buffer format would still need to be designed and implemented for use by user-written subsystems.

Tokens simplify version accommodation and cope with highly variable data.

In the end, it was decided that if the common buffer format could meet the needs of user subsystems, it should meet the needs of Tandem subsystems as well. The cost of converting old subsystem servers to use the new buffer format was offset by eliminating the cost of writing subsystem-specific buffer interfaces. Even requesters benefited from the common format; in addition to the reliability advantages of using a small amount of shared interface code, the common format made debugging the interface much simpler.

Tokenized vs. Structured Data

Next came the question of whether the common buffer format should be tokenized or structured. Tokenized data has the advantage that it is self-describing, making it easy for interpreters to work with many different subsystems in a general way. Specific subsystem knowledge is not required because tokens of data can be manipulated and displayed in ways appropriate to the token type.

Tokenization also simplifies the problem of version accommodation because subsystems can easily detect the presence of an unrecognized token in an up-level request or the absence of an expected token in a down-level request. Finally, tokens are an excellent way to cope with highly variable data such as variable lengths, variable numbers of occurrences, and variable data interpretations and formats. Fully tokenized interfaces have the disadvantage, however, that the sheer quantity of tokens can make the interface both difficult and expensive to use.

Most compiled programs are capable of and comfortable with dealing with data structures containing multiple related fields. For many of these programs, fixed data structures are the most natural and efficient format for storing and manipulating data. Structures have the disadvantage that fields cannot be omitted and can be added only at the end. These problems are commonly surmounted, however, by using null values and Boolean flags to indicate the presence or absence of fields.

Version accommodation for structures is typically handled by allowing new fields to be added only at the end of the structure. Up-level requests are those with a structure length greater than the length expected, and down-level requests are those with a shorter length. Great care must be taken when compiling an old program with new declarations since the old declarations may contain new fields that the old program neither recognizes nor initializes. The main disadvantage of fixed structures is that they cannot accommodate highly variable data.

Instead of using either a fully tokenized or fully structured buffer format, SPI combines their advantages by providing both tokens and structures. A token value may be a single data element, or it may be a data structure containing several fields. Either way, the SPI access procedures manipulate each token as a unit. Simple tokens containing a single data element are used for variable-length data, variable occurrences, parameters to be standardized

across a wide variety of messages, and parameters to be independently interpreted. Structured tokens are used to group related data items and simplify programmatic access to the data.

SPI Buffer Format

Each SPI buffer begins with an SPI-defined header containing standard information such as the overall length of the buffer, the amount of the buffer being used, the subsystem owning the buffer, version information, and position information. The header also contains information specific to the type of message being used. Examples of this include the command number in command messages, the server version in response messages, and the event number in event messages. Although the header is stored as a structure, its fields are made available to SPI users via the tokenized access procedures.

The main part of the SPI buffer contains a sequence of SPI tokens, each consisting of an identifier and an associated value. The SPI procedures allow tokens to be stored and retrieved by identifier; programs using the buffer typically do not need to know either the byte position or the order of tokens within the buffer. Figure 4 provides an overview of the SPI buffer format.

This loose assortment of tokens in the buffer is sufficient for most uses. In some cases, however, a message buffer must contain tokens that are partitioned into sequential or hierarchical sets. A command that refers to multiple objects, for example, might receive a reply containing multiple sets of response information. Different sets of response information in the same buffer may contain variable occurrences of the same tokens; this requires a way to partition sequential sets of tokens. Another example is that of nested error information where a subsystem operation fails because a second subsystem got an error from a third one. Returning all the error information requires a way to partition the associated error tokens hierarchically.

Both of these partitioning requirements are satisfied by an SPI construct called a *list*. A list token is used to start a list, and an end-list token ends it. All the tokens in between are logically grouped together. This is one of the few instances where the order of tokens is important.

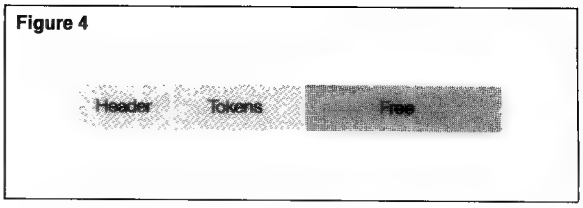


Figure 4.
SPI buffer format. An SPI buffer consists of an SPI-defined header followed by subsystem-defined tokens.

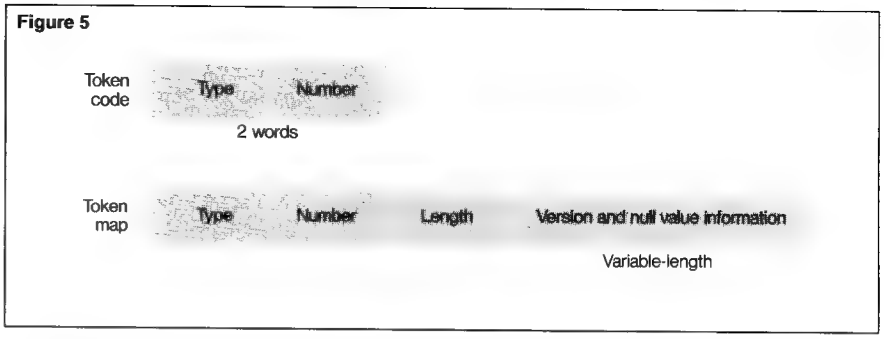


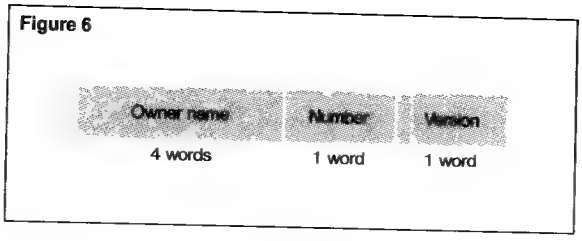
Figure 5.
Token identifiers. A token identifier may be either a token code or a token map.

SPI Token Identifiers

The simplest view of a token identifier is that of a number or code that tells what the token value means. SPI elaborated on this simple concept in order to describe different things to different potential interpreters of tokens. An SPI token identifier may be either a token code or a token map. A token code is a 32-bit value containing a token data type, a token length, and a token number. At the lowest level, within the SPI buffer, every token is identified by a token code.

Token maps describe structured data values that may be extended in later releases. A token map contains a 32-bit token code value augmented with version and null value information about the structure. Since the token map used by a program always corresponds to the structure version used by the program, these structures can be extended without affecting programs written to use the old declarations. Figure 5 compares token codes and token maps. Token maps will be discussed later in connection with version accommodation; the present discussion focuses on simple token codes.

Figure 6.
Subsystem ID. A subsystem ID (SSID) is a 6-word data structure consisting of an 8-character owner name, a 16-bit subsystem number, and a 16-bit version. The version field is primarily for information. The owner name and number are sufficient to uniquely identify a subsystem.



Since data type and length are part of the token code, interpreters and the SPI procedures themselves do not need to have any special knowledge about the subsystem defining the token. SPI uses the data type and length to determine how much data to move and what alignment to use when storing or retrieving token values. Interpreters use this information to determine how to manipulate token values or format them for display. So that they can be used for this purpose, the data type values are defined by SPI and are global to all subsystems. In addition to standard programming language data types such as integer, character, Boolean, and enumerated type, SPI also defines conceptual data types such as time stamp, file name, and transaction ID. Different data types are defined for widely used logical types that require different conversions from internal to external form.

Subsystems must use these SPI-defined data types when defining their tokens. Subsystems distinguish between different tokens by giving them different token numbers. Take, for example, a command that duplicates a GUARDIAN file:

DUPLICATE FILE *source*, *destination*

In converting this to an SPI command buffer, the command code for DUPLICATE and the object type code for FILE would both go in the SPI header, and the *source* and *destination* parameters would both go into the buffer as tokens. Both of these tokens would have the data type for a GUARDIAN file name, allowing an interpreter such as TACL to accept or display the values in file name format. The subsystem receiving the command buffer can distinguish between the source and destination file names by assigning them different token numbers.

Subsystem Identifiers

Aside from a range of numbers set aside for SPI and EMS, subsystems are free to use whatever token numbers they choose. Since the same token numbers are available to all subsystems, the same token code can easily mean different things depending on the subsystem using it. To distinguish between them, each token identifier is qualified by a subsystem identifier, either explicitly or by context.

The SPI subsystem ID was designed to allow customers and third parties to assign their own subsystem ID values with a minimal risk of conflicting with others. Each subsystem ID consists of an owner name, a subsystem number, and a version number, as shown by Figure 6. Each organization assigning its own subsystem IDs may pick an eight-character owner name; this name should be unlikely to conflict with the owner name of any other organization. Tandem, for example, has picked TANDEM as its owner name. This owner name will be used to define subsystem IDs for all subsystems supplied by Tandem.

The owner name and subsystem number alone are sufficient to identify the subsystem. Two subsystem IDs are considered identical if the owner names and subsystem numbers match. The version number is an informational field only. Whenever a new SPI buffer is initialized, the ID of the subsystem defining the buffer is included in the header. The version field in the subsystem ID thus indicates the version of subsystem declarations used by the program that formatted the buffer.

Subsystem Declarations

Making SPI available to multiple programming languages requires generating definitions for data structures, token identifiers, subsystem IDs, and other associated constants in all of those languages. This would be an error-prone and almost impossible task to do by hand. Luckily, however, the entire process has been automated through enhancements to Tandem's Data Definition Language (DDL).

Each subsystem provides its SPI definitions using DDL, and the DDL compiler is then used to build the corresponding definitions in TAL, COBOL85, and TACL. (See Figure 7.) A program can use the SPI interface for that subsystem by including the appropriate subsystem-supplied declarations in its compilation. The SPI access procedures are a resident part of the standard GUARDIAN library and can be invoked by any TAL, COBOL85, or TACL program.

Access Procedures

All SPI messages are built and decoded using the following five access procedures:

- SSINIT initializes an empty message buffer.
- SSNULL sets an extensible structured token to null values.
- SSPUT adds a token to a message buffer.
- SSGET retrieves a token value from a message buffer.
- SSMOVE moves tokens from one buffer to another.

Figure 7

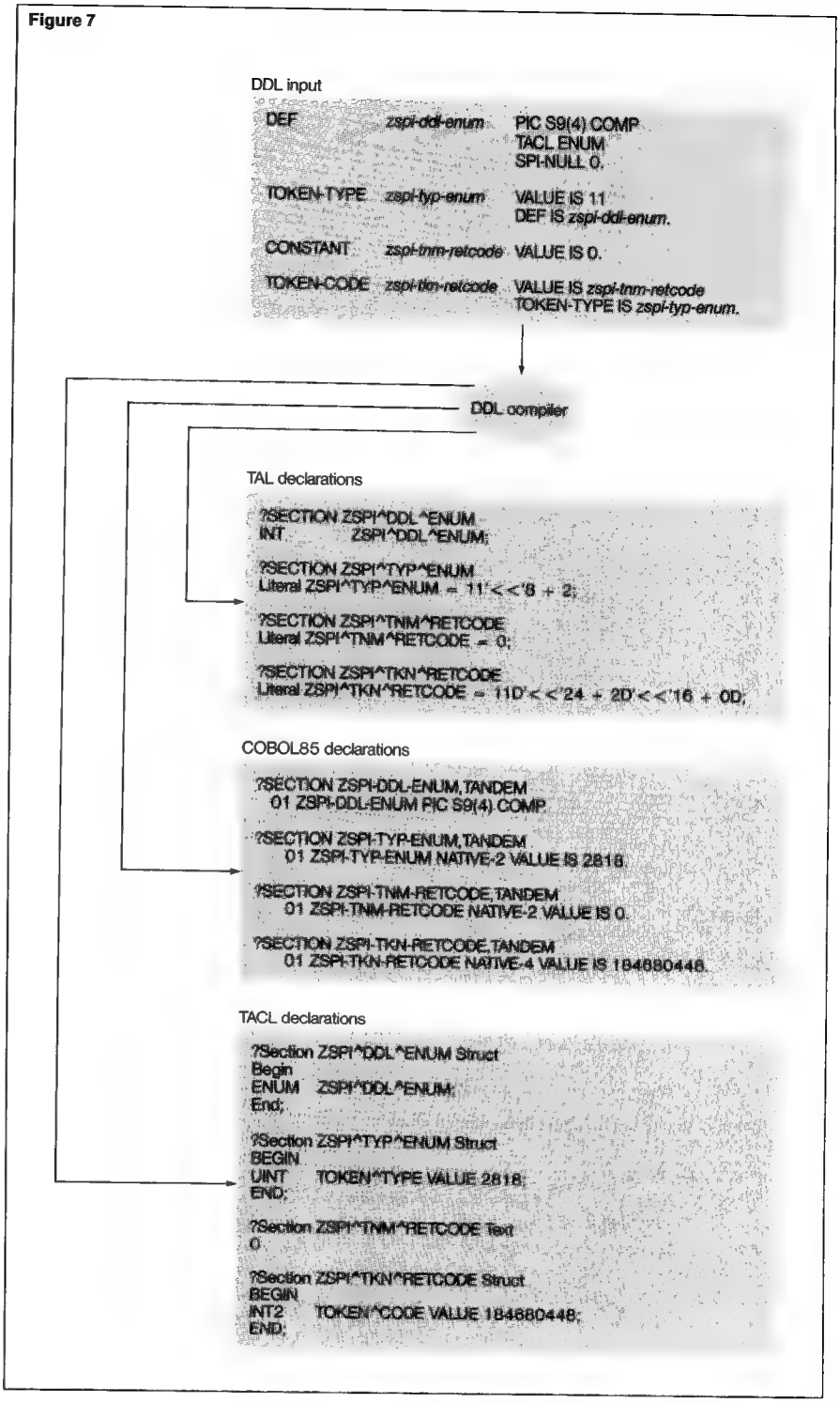


Figure 7.

Use of DDL to generate SPI declarations. Tandem's Data Definition Language (DDL) can automatically translate SPI declarations into TAL, COBOL85, and TACL equivalents.

Figure 8

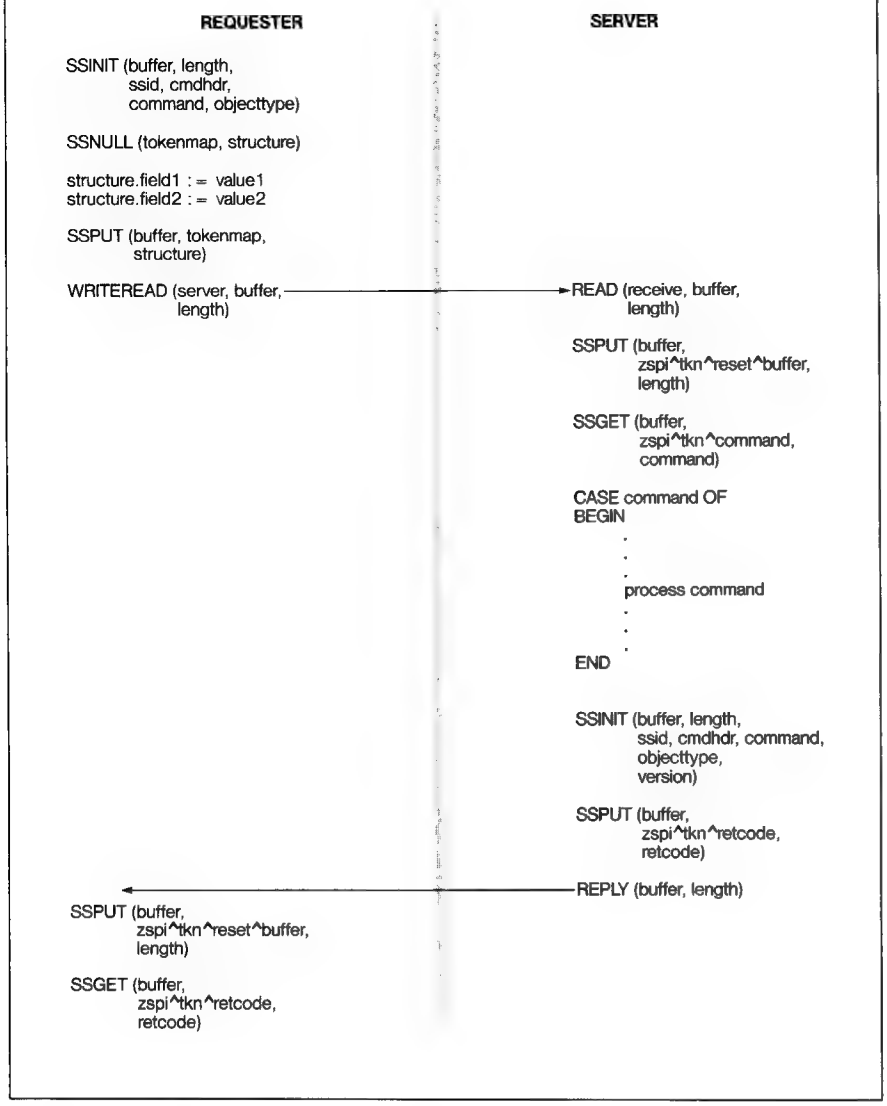


Figure 8.

Access procedures example. This pseudocode shows how the SPI access procedures are used in a simple command-response interaction. The requester uses SSINIT to initialize the buffer, SSNULL to initialize extensible structures, and SSPUT to add tokens to the buffer. Upon receipt of the SPI message, the server adjusts the buffer length using an SSPUT call and then uses SSGET to retrieve information from the buffer. The server uses SSINIT and SSPUT to build the response message, and the requester uses SSGET to retrieve response tokens.

SSINIT initializes a new SPI buffer. It builds the SPI header using information such as the buffer length, subsystem ID, and buffer type. Some SSINIT parameters depend on the type of buffer being built. For example, command and object type numbers are accepted for command buffers, and event numbers are accepted for event-message buffers.

SSNULL initializes structured token values that are identified by token maps. The token map contains information about the appropriate null value to be used for each field. Once the structure has been initialized with null values, the fields of interest can be filled in by the program. The null values in the unused fields will tell the recipient of the SPI buffer that the fields are not supplied.

SSPUT accepts a token identifier and token value, adding the token to the end of the SPI buffer. It also accepts special SPI tokens that allow the caller to perform control operations such as deleting tokens or re-initializing the current SSGET position.

SSGET accepts a token identifier and returns the associated value. By default, SSGET starts searching for the token starting from its current position. As each token is found, the current position is advanced so successive SSGET calls may be used to extract successive occurrences of a token. If an explicit index is specified, SSGET will start over at the beginning and return the specified occurrence. SSGET also accepts special SPI tokens that allow the caller to perform operations such as scanning the buffer token by token or obtaining the length or number of occurrences of a specified token.

SSMOVE gets tokens from one buffer and puts them into another buffer. This access procedure is not really necessary because any SSMOVE operation can be performed as a succession of SSGET and SSPUT calls. The SSMOVE procedure is provided as a convenience.

The same procedures are used by both message producers and message consumers, greatly simplifying the task of programs that must act as both. This is illustrated in the simple command-response example shown in Figure 8.

SPI Standards

The SPI declarations and access procedures provide the raw materials for building subsystem programmatic interfaces. The SPI standards provide the blueprints needed to make sure that all these interfaces are built the same way. The standards are layered and extensible so that customer and third-party subsystems can add their own standards and usage rules to those already defined by Tandem. Figure 9 shows the Tandem standards that have been layered on SPI and illustrates how customer-defined standards may be added.

SPI Command Language Standard

Within Tandem's DSM architecture, command and response interfaces are governed by Tandem's SPI Command Language Standard (CLS). The SPI CLS is a Tandem internal document that specifies naming conventions for SPI declarations and rules for command/response interactions, command continuation, response information for multiple objects, and error descriptions. The portions of the standard that apply to user-written subsystems are documented in the *Distributed Systems Management (DSM) Programming Manual*.

Naming Rules

SPI naming rules were formulated so that individual organizations and subsystem developers could independently invent their own names and still be protected from name conflicts. All SPI declarations are given names of the form:

subsys-type-name

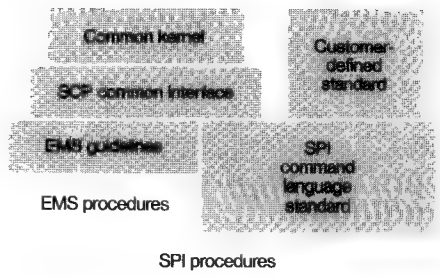
where

subsys is a four-character subsystem abbreviation. Tandem subsystems reserve all abbreviations beginning with the letter Z.

type is a three-character mnemonic that identifies the type of declaration. Among the mnemonics defined by the CLS are:

CMD	Command number
OBJ	Object type number
EVT	Event number
DDL	Structure definition
TKN	Token code
MAP	Token map
ERR	Error number
VAL	Token value

Figure 9



name is a meaningful name selected by the subsystem developer.

Example names in this form include:

- ZPWY-VAL-SSID. The subsystem ID value for PATHWAY
- ZSPI-TKN-RETCODE. The token code for the SPI standard return code token.
- ZCOM-OBJ-PROCESS. The SCP common object type for process.

The first abbreviation in the name allows each subsystem to define its own name without conflicting with other subsystems. Tandem subsystem abbreviations will always start with the letter Z, so that users can avoid conflicts by starting their abbreviations with other letters. Also, the three-letter mnemonic for declaration type makes it easy to tell what kind of declaration is being used, helping to reduce programming errors when using SPI.

Figure 9.
Layered SPI standards. This diagram illustrates how standards and procedural interfaces may be layered on top of SPI. Standards may be independent of other standards at the same level or may be built upon standards at a lower level. Naming rules and predefined value ranges allow customers to define their own SPI standards without conflicting with the standards defined by Tandem.

The SPI CLS also specifies rules for field names because some languages, like COBOL85, allow unqualified field names to default. The field name rules prevent a name conflict from inadvertently making a customer-supplied field inaccessible. For example, suppose SPI defined a data type that included a field named CPU. If a customer used the SPI data type within a structure of his own that was also named CPU, there would be no way to reference CPU unambiguously.

Users should avoid conflict with Tandem structure declarations by not using field names that begin with Z. Tandem subsystems still need protection, however, from other Tandem subsystems in the cases where declarations are shared. So within Tandem, shared declarations use field names that begin with Z- (i.e., Z-*name*), and private declarations use field names that begin with only Z (i.e., Z*name*).

Command-Response Interactions

The SPI CLS sets guidelines for how SPI servers should be opened; how commands, objects, and responses should be defined; and how errors should be returned. The CLS stresses the use of simple methods that will do the job for the majority of subsystems; this discourages subsystems from inventing complicated and contradictory special cases.

One important simplifying rule is that the subsystem must specify a buffer size large enough to hold all the tokens necessary for a single command or the response information about a single object. This removes the need for special mechanisms that allow commands to be continued over multiple buffers and sent in separate messages. Since a subsystem either receives the entire command or it doesn't receive the command at all, partial commands do not exist. Similarly, each response message must contain one or more complete response records, each of which provides all the response information about a single object.

Another simplification is that the use of multiple commands per buffer is prohibited. Each buffer contains only one command; this is much simpler for the servers since they only need to respond to one command at a time. Multiple commands must be issued as multiple messages, using one command per message.

In the simplest case, a single-command message results in a single-response message containing a single-response record. This simple sequence will suffice for the vast majority of situations. For the few cases not covered by this simple model, the CLS defines optional extensions that may be used by the subsystems requiring extra features. Those subsystems that do not need or want the extra functionality do not need to deal with the extra complexity. For those subsystems that do need extra functionality, the CLS provides a standard way to obtain it.

Multiple response records are a case in point. A single response record contains all the response information about a single object. If a subsystem defines a command that operates on multiple objects, perhaps via wild cards in the object name, there needs to be a way to handle multiple response records for a single command. This requires at least a mechanism for continuing requests over multiple response buffers, using one response record per buffer. In the general case, it is desirable to permit multiple response buffers with multiple response records per buffer. The CLS specifies mechanisms for all these cases.

The simplest applications need not worry about continuation or multiple response records at all. The CLS permits subsystems to support only commands that operate on single objects. Such a command will only generate a single response record, so continuation and multiple response record mechanisms are unneeded.

Subsystems that need or want to support multiple-object commands, such as INFO *, must return a context token in each response buffer until all response records have been returned. To continue to receive response records, the requester must resubmit the command with the context token. The context token will tell the server how to continue the response at the point where the previous response buffer left off. For commands that operate on a large number of objects and generate a small amount of response information, it may be desirable to pack multiple response records per buffer. This requires using the SPI data list construct to separate the different response records within the buffer.

This last issue (the use of data lists to delimit response records within a single buffer) illustrates the CLS philosophy of keeping simple things simple and placing the burden of added complexity upon the users of more complex features. Requesters need never be concerned with data lists; they are unneeded in the single-response-record case, and single-response records are the default. A requester that asks for multiple response records will need to deal with moving from one list to the next as it processes the response records.

Another CLS principle is that, whenever possible, it is better to deal with special cases once in the server than to force each requester to deal with special cases independently. For example, when a requester asks for multiple response records per buffer but the response buffer contains only a single-response record, the CLS specifies that the response record must always be enclosed in a list. Although this complicates the job of servers that only return single-response records, it relieves the requester from having to supply special-case code to handle them. Requesters asking for multiple response records can expect each of them to be enclosed within a list, regardless of how many are returned in each buffer.

Errors and Error Lists

The SPI CLS recognizes the need for two kinds of error information. A simple program attempting to execute a command needs a simple single error code that can be used to decide whether to continue, retry the operation, or report an error. A more sophisticated program (or a human operator evaluating an error message) needs more detailed information about how and why an error has occurred, including perhaps the description of other errors that caused or contributed to the failure.

The CLS handles the first requirement by requiring each response to include a standard ZSPI-TKN-RETCODE token containing a subsystem-defined error code. All subsystems use a return code value of 0 to indicate successful completion.

The SPI CLS emphasizes simplicity in command-response interactions.

The second requirement is handled by encapsulating the related error information in error lists. Error lists may be repeated or nested and may contain any number of tokens. This fits the error model very well because a single command may encounter multiple warnings or errors, and errors encountered by lower levels of software may be passed along to higher ones. This detailed information in an error list can be interpreted by a program, passed along to a higher level, included in an event message, or formatted in a display message for a human operator.

Another advantage of error lists is that they provide a convenient subsystem context for the tokens that they contain. Unless a subsystem qualifier is explicitly specified, tokens within a buffer are assumed to be qualified by the subsystem ID used to initialize the buffer. A PATHWAY buffer is expected to contain PATHWAY tokens, a TMF buffer is expected to contain TMF tokens, etc. If PATHWAY, while processing a PATHWAY command, sends an SPI request to TMF and receives an error, the PATHWAY response will contain a PATHWAY error list, with PATHWAY tokens describing the error. Nested within that list will be a TMF error list containing TMF tokens that describe the TMF error. All tokens within a list are qualified by the subsystem ID of that list, and list tokens moved from a TMF buffer to a PATHWAY buffer remain qualified by TMF. The use of error lists to report nested errors in response messages can be easily applied to reporting nested errors in event messages.

Error lists and event messages, containing variable numbers and types of tokens, can be interpreted by text formatters to produce parameterized error messages. EMS provides a text formatter that, given an SPI event buffer and an ASCII text template, will produce a formatted ASCII text message. This has obvious extensions for national language support: different text templates can be defined for each supported language.

EMS Guidelines

Just as the SPI CLS provides standards for command and response messages, the EMS guidelines provide standards for event reporting. The EMS guidelines describe what events to report, how to report them, what information to include about them, and how to document the associated event messages. Standard tokens are defined for event information such as the event number, the message generator, event emphasis in displays, compatibility with the old operator messages, and associated event text.

EMS designers faced an interesting challenge in finding a standard way to designate the subject of an event message. Each event message describes or is associated with some subsystem component, either a hardware component, such as a controller or a device, or a software component, such as a process or a protocol layer. This component is the subject of the event message. Subjects and the information needed to describe them vary so much from subsystem to subsystem that it is difficult to define a single standard subject token that can describe them all.

EMS solved this problem by defining a subject-mark token, ZEMS-TKN-SUBJECT-MARK, indicating that the next token in the buffer describes the event subject. This is one of the few position-dependent tokens used with SPI. It accomplishes the dual goal of allowing subsystems to define their own subject tokens and giving event-message consumers a standard way to find the subject of any event message.

SCP Common Interface

The SPI CLS and EMS guidelines define the basic framework upon which additional standards can be built. Many Tandem subsystems use the Subsystem Control Point (SCP) process as the gateway to their programmatic interface. In the case where multiple management programs must control and monitor multiple subsystems, having the operator open each subsystem becomes an $n \times n$ connection problem. Instead, all management programs can open the SCP process and gain access to all cooperating subsystem servers. This provides full connectivity with an $n + n$ solution.

To present an efficient and uniform interface through SCP, SCP common interface standards were developed to provide a common core of commands, object types, errors, and events. The SCP common interface is a layer on top of the SPI CLS and EMS guidelines providing further standards in areas that they leave open. For example, the SPI CLS specifies that names will be of the form *subsys-type-name*, but leaves the selection of *name* to the subsystem designer. The SCP common interface provides a concordance of names so that all SCP-related subsystems will use the same token names for the same object types. Without the SCP common interface, different subsystems might define a program token using the names -PROGRAM, -PGM, and -PROG. Instead, all SCP-related subsystems have standardized on -PROG.

Commands, object types, tokens, and events defined by the common interface use ZCOM as the subsystem prefix and are assigned within a reserved range of the numbers available for general use. This prevents conflicts with names and numbers assigned by subsystems using SCP.

Further standards can easily be layered on top of these. Tandem's SNAX and MULTILAN™ products use a shared kernel of common code to provide DSM functions including SPI and EMS support. This common kernel uses the prefix ZCMK to identify its shared tokens, errors, and events, which are layered on top of the SCP common interface standards to provide even greater uniformity. By using a unique name prefix and selecting token values within a predefined range, the common kernel avoids conflicting with the names and values chosen by other subsystems.

Version Accommodation

Version accommodation is necessary because interfaces change in response to user requirements. New subsystems are added, new features and functions are defined, and obsolete functions and features are dropped. To adapt to future requirements, SPI must permit interface changes. Programs that take advantage of new features can and should be expected to change. But for a distributed environment to remain manageable, old programs that do not use new features should neither need nor be expected to change. SPI version accommodation allows subsystem interfaces to continually change, but protects the programs using those interfaces from having to continually change with them.

The subsystem ID used to initialize an SPI buffer provides the version of the declarations used to format the buffer. This can be used as a rough guide but should not be used as the basis for rejecting incompatible messages. A message formatted by a later version may still be compatible with earlier versions. Since all the information in an SPI buffer is contained in tokens, SPI version accommodation goes down to the token level. The following discussion describes how version accommodation takes place for simple tokens, structured tokens, and extensible structured tokens.

SCP guidelines add another layer of standards on SPI, CLS, and EMS.

Simple Tokens

Simple tokens are either present or not, and a simple accommodation scheme can be based upon this fact. A message consumer interested in particular tokens, such as a management program examining a response buffer or an event message, can simply check for the presence and the validity of the desired tokens. If an expected token is absent, the message generator may be at a version level that does not support the token. Similarly, if a token value exceeds the expected range, the message generator may be at a version level that supports an extended range. In addition, every response message contains a server-version token that gives the server's version and can be used as an additional indication of the version of the response.

A message consumer that must understand all the information in a buffer, such as a subsystem server receiving a command buffer, must detect version incompatibilities by examining all the tokens in the buffer. A server that finds an unexpected token or an invalid token value in a command should reject the command with an error. This takes care of both the case in which the requester has made a genuine error and the case in which there is a version incompatibility. On the other hand, if all tokens and values are understood by the server, the command can safely be processed regardless of version differences between the requester and server.

Structured Tokens

Version accommodation for structured tokens can be handled in the same way as for simple tokens, as long as the structure itself is never changed. This method is applied to tokens using structured data types such as those defined by SPI for file names, transaction IDs, and subsystem IDs. These data types may be incorporated into subsystem-defined data structures and can never be lengthened or shortened since doing so would change the offsets within fixed structures incorporating these types. The simple token methods work well with tokens based on these simple structured types.

Extensible Structured Tokens

In general, however, the version implications of structured tokens are more complicated. If a structure contains fields that are not always used, the presence or absence of the structured token cannot be used to indicate which fields have been supplied. Fields are defined by their byte offsets within the structure so the length and placement of existing fields cannot be changed. Additional fields may be added to the end of structure without interfering with previously defined fields, but this increases the structure's overall length.

The newly extended structure will not fit within the shorter structures allocated by programs compiled with old structure definitions. If an old program gets compiled with a newly extended structure definition, it will acquire new fields that it does not use and does not properly initialize with null values. SPI solves these problems by providing structured tokens that are extensible; that is, they may contain fields that are not always supplied, and they may be extended by adding fields to the end of the structure.

SPI supports extensible structured tokens by restricting the ways in which these structures may be used, by offering an automated method for setting structures to null values, and by automatically adjusting structured token values to fit the declarations being used. The restriction is that extensible structures may be extended only by adding new fields to the end of the structure. Each extensible structure is a token by itself. It cannot be used as a subcomponent of another structure since doing so would prevent it from being extended in a later version.

Each field of an extensible structure must have a defined version and a null byte value. The version specifies the minimum software level required in order to understand that field. The null byte value, if repeated throughout the entire field, indicates that no value has been supplied and the field should be ignored. Note that the null value is not the same thing as a default value (though, in fact, the actual values may be the same). The null value indicates that a field has not been supplied; the default value is the value to be used or assumed when a null value is found. SPI null byte information is associated with each field of a data structure when the structure is defined using DDL. Version information is associated with the token map when the token map is defined using DDL. The DDL compiler converts these DDL definitions into corresponding source-language definitions for both the data structure and the token map.

Version information must be associated with the token map definition, not the data structure definition, because tokens introduced in different releases may have values based on the same data structure. For example, the DUPLICATE command might be defined with a SOURCE token map and a DESTINATION token map, both associated with the same data structure. A later release may define an additional ERRORFILE token map for the command. The token map might be based on the same data structure but would have a later version associated with its fields.

Some fields cannot be assigned a null value because all possible values are meaningful. These fields are defined with "no version" since the content of the field cannot be used to determine whether or not the field has been supplied. The actual version of the field is assigned to a separate IS-PRESENT field that indicates whether or not the field of interest is present. Assigning a null value of false to the IS-PRESENT field provides exactly the right meaning needed for version accommodation. If the field is supplied, the IS-PRESENT field will be non-null, and the field version will apply. If the field is not supplied, the IS-PRESENT field will be null, and the field version will not be used in version calculations.

Use of Token Maps

The token map allows the SSNULL procedure to automatically set all the fields of a structure to null values. This solves the problem of an old program getting compiled with new extended definitions. The new definitions may contain structures with new fields, but the new token maps will contain the corresponding null value information. When SSNULL is called, it will correctly initialize those fields with null values. The entire extended structure may then be placed in an SPI buffer with no danger that the new fields will be interpreted as containing meaningful values.

The token map allows the SSPUT procedure to automatically calculate the maximum field version among all the extensible structures in the buffer. For example, a structure may contain fields defined with versions of the C00 and C10 releases. If only the C00 fields contain non-null values, the maximum field version is C00, not C10.

SPI version accommodation covers simple tokens, structured tokens, and extensible structured tokens.

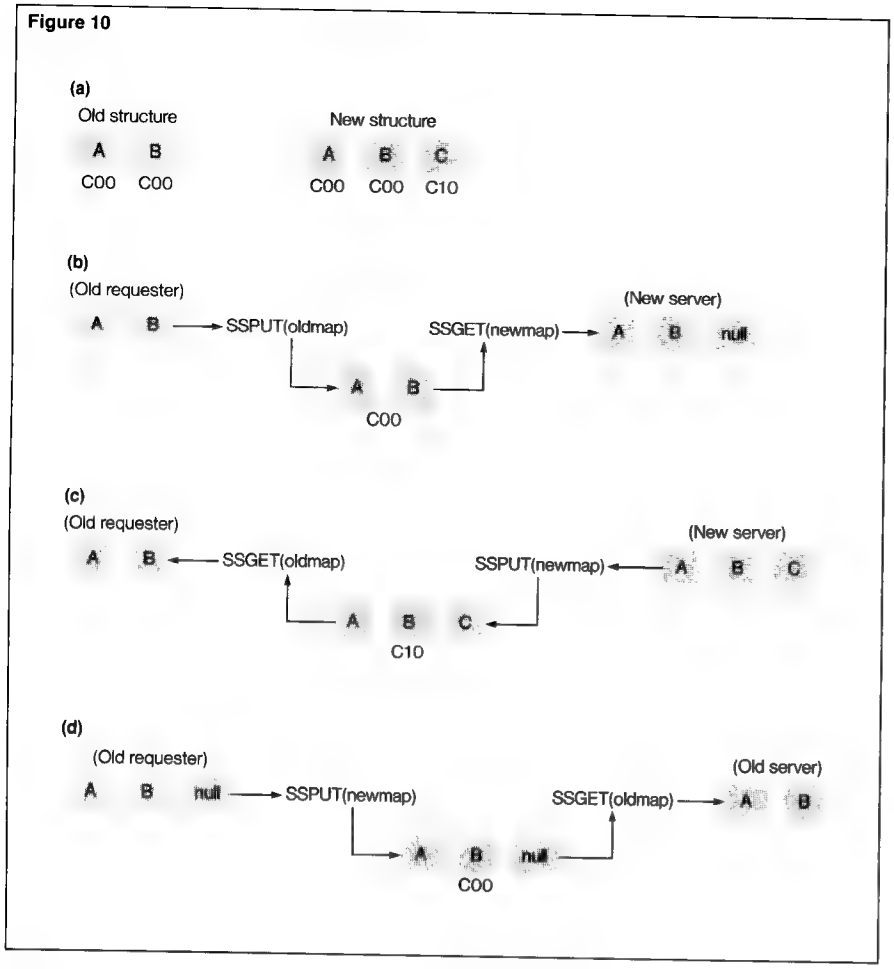


Figure 10.
SPI will automatically adjust for programs using different declaration versions. (a) Version accommodation. (b) Old version to new version. (c) New version to old version. (d) Old version compiled with new declarations.

SPI updates the maximum field value as each extensible structure is added to the buffer. Subsystem servers examine the maximum field version and use it to reject requests containing later-version fields. The requester is not burdened with the complexity of sending old structures to old servers and new structures to new ones. The same declarations can be used with all servers, but the new fields can only be used with the servers that support them. This makes the use and detection of new fields in structures as simple as the use and detection of new tokens.

Finally, token maps allow the SSGET procedure to automatically adjust extensible structures to the size expected by the caller. If the structure in the buffer is longer than the expected length indicated by the token map, the value returned by SSGET is truncated to fit. If the structure in the buffer is shorter than expected, the token map is used to return a value in which the extra fields expected by the caller have been set to null values. As shown by Figure 10, programs using different declaration versions can send structures back and forth, and SPI will automatically adjust them according to what is expected at either end. This eliminates the need for any special-case structure conversion code.

The example shown in Figure 10a illustrates how SPI accommodates the addition of a new structure field in a later release.

If the buffer contains an old version of the structure, SSGET uses the null value information in the new token map to fill the missing fields with appropriate null values. (See Figure 10b.)

When a new version of the structure is added to the buffer, SSPUT uses the version information in the new token map to set the buffer's maximum field version, which can be used by down-level programs to detect that the buffer contains an unrecognized field. When a new version of the structure is retrieved with an old token map, SSGET truncates the structure value to the old length. (See Figure 10c.)

If an old program is recompiled with new declarations, it will start using the new version of the structure, but none of the code within the program will be aware of the new field. SSNULL (which is called by the program to initialize the structure) will use the token map information to set the new field to null values. When the structure is added to the buffer, SSPUT will set the maximum field version based on the old version because only the old fields contain non-null values. The resulting buffer will be accepted by old servers because even though it contains the new version of the structure only the old fields are used. (See Figure 10d.)

SPI CLS Version Rules

The SPI version accommodation features can be used to satisfy the CLS guidelines for version compatibility. The CLS specifies that servers must work with all previous requesters and must be able to support the current level of functionality for all future requesters. For management programs/requesters, this means that they will continue to work with future versions of servers and declarations, and that they will be able to use all the functions supported by earlier server versions.

One constraint on the use of SPI version accommodation is that of the overall SPI buffer size. New extended structures must fit within the message buffers used by older servers to receive messages. The CLS requires subsystems to define minimum buffer sizes that allow for future growth and also specifies ways in which the defined buffer sizes may be compatibility extended.

Conclusion

SPI unifies the highly diverse DSM environment—different applications and subsystems, written in different programming languages, using different I/O mechanisms between different versions of software at different locations—by providing a standard way to build and decode the messages used for commands, responses, and events.

To accommodate this diversity with a single mechanism, the SPI design uses:

- A small number of simple primitives for building and decoding messages.
- A tokenized binary message format that also supports structures and hierarchies.
- A layered set of interface standards to prevent conflicts and provide uniformity.
- An automatic version accommodation scheme that provides operability across multiple versions with a minimum of conversion code.

These SPI design elements solve the many programming problems caused by using other interfaces that were text-oriented, nonstandard, version-dependent, language-specific, and otherwise ill suited for DSM.

With SPI, the techniques used in one DSM program can be easily applied to other languages and other subsystems, easily used with older and newer program and subsystem versions, and easily extended as new languages and subsystems are introduced in the future.

References

Distributed Systems Management (DSM) Programming Manual. Part no. 82587. Tandem Computers Incorporated.

Acknowledgments

The author wishes to thank all the people who contributed to the design and definition of SPI, including Richard Carr, Ross Yakulis, Bernice Malizia, Pete Homan, Phil Garrett, Paul Robinson, Jonathan Sechrist, Marc Desgrousilliers, Keith Stobie, Bob Strand, Michael Stewart, Randy McRee, and many others. Special thanks go to Christy Scharf and Tom Eastep for their careful review of this article and many constructive comments.

Gary Tom joined Tandem in 1978 and worked in Software Education and Software Support before transferring to Software Development in 1981. He is currently a member of the Operating Systems Group in Tandem's Transaction Networks Division. He has a bachelor's degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology.

Event Management Service Design and Implementation

Event Management Service (EMS) is the event processing product in Tandem's Distributed Systems Management (DSM) environment. EMS allows Tandem and customer subsystems within a GUARDIAN 90™ system to define, report, retrieve, select, and display management data.

This article discusses the evolution and architecture of EMS, the generation and use of tokenized management data, and the collection and distribution of management data.

Evolution of EMS

Certain information generated by subsystems is of interest to various groups of network users. This type of information is called management data.

Management data describes diverse situations, both positive and negative. It is distinct from user data, which individual users or groups control and create. Any system component can create or report it, and it is frequently generated unexpectedly. Because the user of the component generating this information is not always the user that needs it, a system operator process acting as a central collection point is required.

In Tandem networks before DSM, this function is provided by a system operator process called $0. System and application processes report management data by sending messages to $0, which then adds header information (a timestamp and originator identification) to the message. The combination of header information and management data is called an operator message. This operator message is sent to $AOPR, saved in the OPRLOG, and displayed on a console device.

When using $0 in large pre-DSM systems, a variety of problems are encountered including performance limitations and restricted operator message distribution (messages can be sent to only one terminal, disk file, and single user process). In addition, distributing and processing text messages is problematic. These limitations, in combination with customer-requested improvements, prompted the development of EMS.

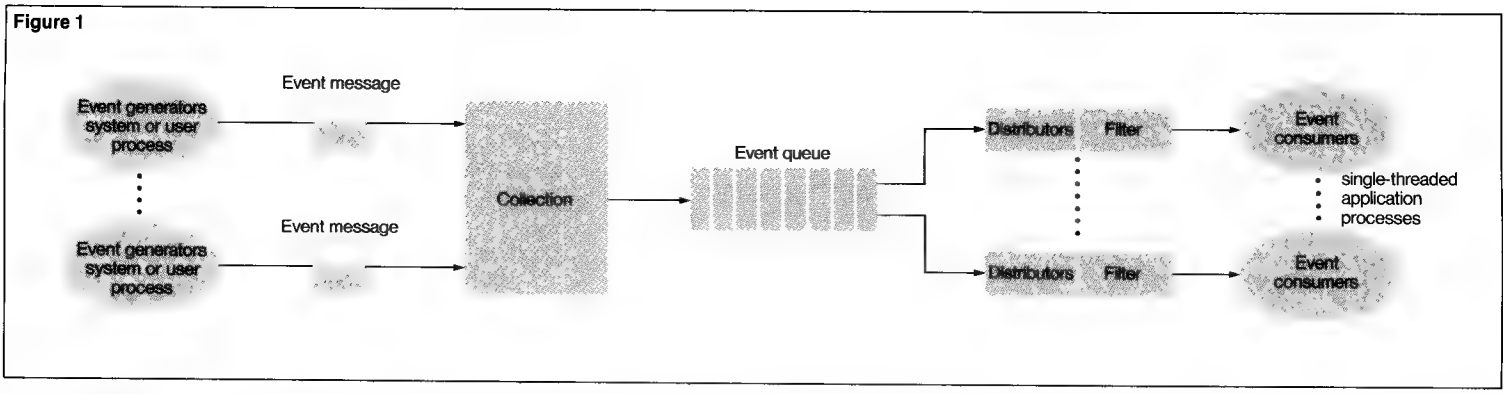


Figure 1.
EMS conceptual sketch. Note that the event message is the subsystem ID event number plus subsystem-specific information.

Design Requirements

Establishing the design of EMS was a multistep process. First and foremost came the list of customer requirements. Taken as a group, these requirements pointed to a more flexible and functional system operator process, one that would meet a wide range of customer needs. A more adaptable message format and message construction was requested. The new EMS should provide a single collection point for both Tandem and customer event messages and allow distribution of unique subsets of events to many destinations. Selective message processing by multiple consumers was also a key requirement. From these requirements, certain functions were identified that would support this type of functionality. EMS should:

- Support a two-part message format; the first part containing standard information, the second part containing information tailored to the event.
- Allow detailed messages with generous length limits.
- Use disk-resident log files to store management data; allow these files to be relocated to volumes other than the system disk.
- Allow real-time and historical access to messages.
- Use filters in message distribution to allow event consumers to select a specific set of events.
- Provide a variety of event distributors to match the needs of event consumers.

EMS Design

With the design of EMS narrowed down to the list of general functions, a conceptual view of EMS emerged. (See Figure 1.)

The design of EMS is built on five primary concepts:

- Report, save, and distribute management data as tokenized events rather than as text strings.
- Provide a point of collection for management data.
- Provide disk log file queuing to allow independent rates of event collection and distribution.
- Allow multiple event distributors to be configured.
- Ensure compatibility with the pre-DSM system operator process.

The design of EMS involves a number of new reporting and distribution techniques. EMS introduces the use of tokenized event message formats, log files, multiple distributors, and filters. The combination of these concepts and components, among others, results in an event message system that meets customer needs.

EMS Architecture

The primary components of EMS are $0, the distributors, and the log files. (See Figure 2.) The primary collector, $0, accepts and writes management data to disk log files. Using $0 as the primary collector retains the same destination as in the pre-DSM system operator process. When migrating to a GUARDIAN™ release that supports EMS, existing applications do not have to change the destination for operator messages. The Compatibility Distributor, $Z0, maintains the external distribution interfaces of the pre-DSM $0. The other three distributors have the same basic function of reading event messages from the collector's logs, selecting events based on an installed filter, and distributing the selected events to their clients.

Primary Collector

The primary collector accepts management data in the pre-DSM message format as well as in the tokenized event message format. In the case of a pre-DSM "WRITE to $0" or a system-level operator message, $0 converts the information into a tokenized EMS event.

Tokenized event messages are accepted by EMS in two ways: via WRITEREAD from application-level procedures, and via an internally defined message from system-level procedures. This event message interface is used by system-level procedures that cannot use the file system.

In both cases, header information is added by $0 to the event. The tokenized event is then written to the EMS log file.

The EMS collector is configured as a NonStop™ process pair. In some failure situations, event information could be the alternative or adjunct to CPU dumps for determining what happened on a system just prior to a failure. As long as a system continues to function, the primary collector must function. For these reasons, the primary collector's primary and backup CPUs are the same CPUs used to access the system disk.

Disk Log Files

After the header information is added, the event message is sent to an entry-sequenced log file. EMS may be configured to have as many log files as disk capacity allows. As each log file is filled, the collector chains the successive files together using backward pointers. The first event message in each log file, called the FILESWITCH message, contains the name of the previous log file, creating a backward pointer. A backward pointer is also used in situations where a log file becomes unavailable. Logging reverts to a default subvolume on $SYSTEM and continues. The first event that is written to the new log file is a FILESWITCH event. This creates a chain of log files, which allows the distributors to position to the proper log file.

These log files allow collection and distribution to function independently. When events are generated at a high rate, event distribution can fall well behind event collection. The difference is limited only by the amount of logging disk space available. Event generators can operate at their own rate, without waiting for event consumers.

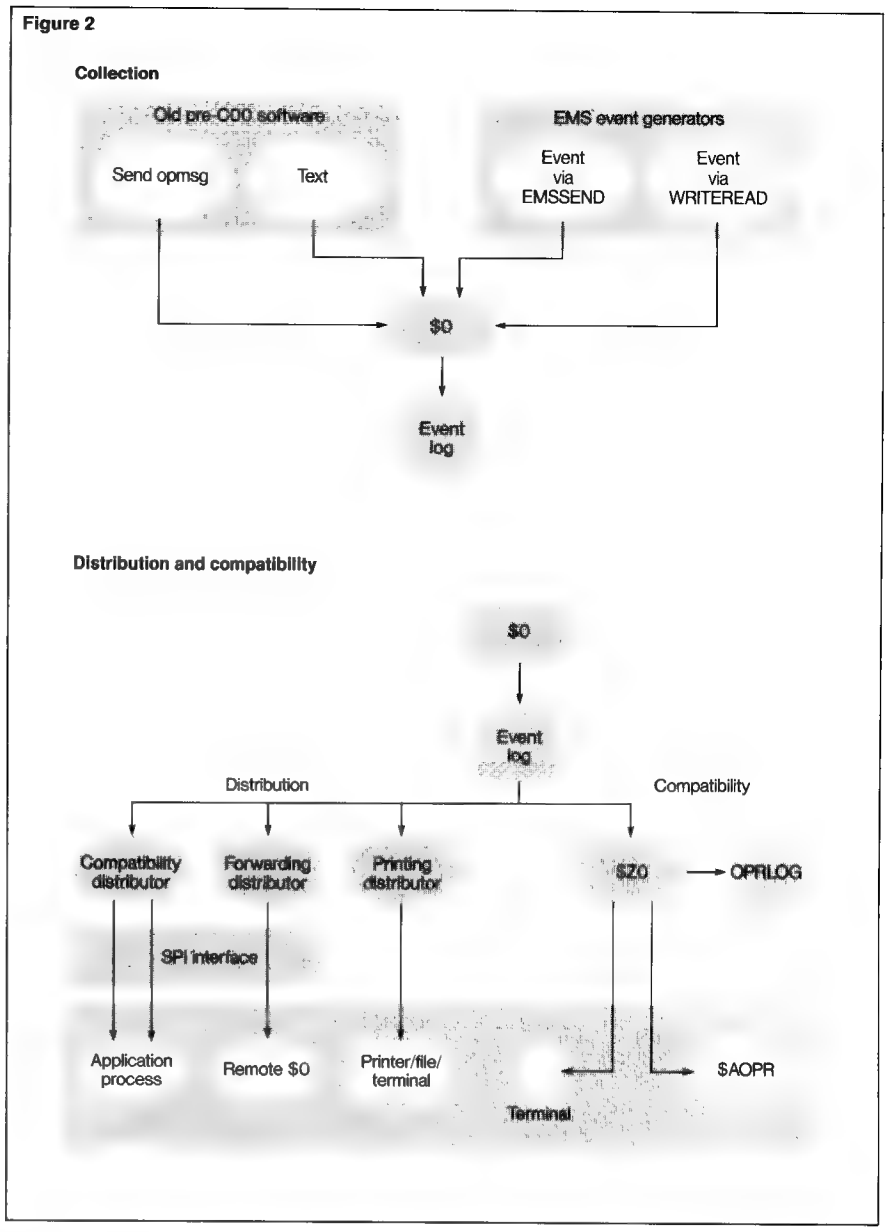
To increase capacity and improve performance, logging can be relocated to another subvolume on another active disk after a cold load. (Locating the EMS log files on the $SYSTEM disk at cold load is necessary to ensure logging begins as soon as the first CPU is loaded.)

Multiple Distributors

Distributors read event messages and deliver them to a destination. A user may configure only one Compatibility Distributor, but as many Consumer, Forwarding, or Printing Distributors as needed.

All of the distributors can retrieve events from active collector/log file combinations. The Compatibility Distributor supports only one collector, while the others may support multiple collectors. The Printing, Forwarding, and Consumer Distributors are architecturally different from $Z0. These three distributors can retrieve events from log files and are capable of processing commands submitted via programmatic interface. They differ only in the type of destination and in the manner the events are delivered (formatted, unformatted; solicited, unsolicited). All the distributors but $Z0 filter messages prior to distribution.

The Compatibility Distributor. The Compatibility Distributor, also known as $Z0, distributes pre-DSM operator messages. It ensures that the same operator message that was used in the pre-DSM system can be recreated in the EMS environment. Since all messages in the event logs are tokenized, $Z0 must be able to build operator messages from tokenized events. For events that have the CONSOLE-PRINT token set, $Z0 calls the text formatter, EMSTEXT. The tokens are decoded by EMSTEXT to produce an operator message text string. The text is the same text that would have existed for the pre-DSM operator message. Then, operating much like the pre-DSM $0, $Z0 writes the operator messages to CONSOLE, $AOPR, and OPRLOG.



The Printing Distributor. The Printing Distributor provides text copies of events to a printer, file, or terminal. This gives users an opportunity to inspect event logs, install filters, and position by event generation time without writing an application. The Printing Distributor allows quick access to log file information and flexibility in defining selection criteria through startup parameters.

Figure 2.
EMS architecture.

The Forwarding Distributor. The Forwarding Distributor funnels messages from a source system to a target system. The Forwarding Distributor selects events from a local collector or log file and sends them to a collector on a remote node. Filtering in the distributor prior to sending events to the remote node reduces event message traffic in the network. The distributor includes a programmatic interface which allows the operating environment to be altered (for instance, changing filters). This type of distributor is useful if events are collected from several nodes and processed at a central site.

The Consumer Distributor. The Consumer Distributor meets the need for event retrieval required by services such as problem tracking and VIEWPOINT™ operations console facility. The Consumer Distributor supports a programmatic interface for event retrieval and distributor control. Events are delivered at the rate of the application program's requests, one event per request. Since the application can directly examine the contents of events, it can react by changing the filter criteria, for instance. A program can start a Consumer Distributor with or without submitting any configuration parameters; the programmatic interface can be used to send configuration commands once the distributor has been started.

Using Tokenized Message Formats

EMS reports, saves, and distributes event messages, allowing consumers to retrieve them as desired. An event message informs operators or programs of an event within a subsystem that may affect system operation. EMS does not dictate the format of the event messages, but the format is important to both the event generators and the event consumers since they create and decipher event messages. To simplify communication between event consumers, distributors, and generators, EMS uses tokens to represent units of information.

A uniform message format is important when developing messages. Uniformity reduces the burden on event consumers who process event messages. Certain information, such as the identity of the process generating the event, is useful in all cases and can be uniformly represented (in this case as a GUARDIAN CRTPID). But soon the diversity of the subsystem environment makes the creation of tokens that have common meaning across subsystems extremely difficult. EMS allows subsystems to define and add their own tokens to events. Using the subsystem ID and event number, the consumer can anticipate the tokens in the remainder of the message. These tokens are generally defined by the subsystem.

Guidelines listed in the *Event Management Service Manual* recommend when to generate event messages and, in broad terms, the type of management data to include. In general, generating and sending an event message to EMS is appropriate when the event generator does not know which operator or program needs the management data being provided. By handing the event message off to EMS, the generator has created a description of the problem and provided notification of that problem.

Independence of Event Messages. All event messages must be easily understood by event consumers. Including all pertinent information about the problem into a single event message is the most reliable way of ensuring that the message is understood and acted upon by its consumers. An event that can't be understood or translated by the consumer is of little use. An example of improper use is a message that refers to an earlier event. This requires the consumer to search for some specific previous event. Such a search can be not only difficult (the consumer would need to know many parameters about the previous event such as when and where it was logged, and have some way of distinguishing it from similar events), but impossible if, for example, the log file that contained the earlier event had been purged.

Event Message Formats. Event messages contain two parts: header tokens and body tokens. The header tokens represent the generation timestamp, information about the event's creator, the type of event (critical, noncritical), and an event number. The body tokens contain information specific to the event.

Required tokens include all header tokens and those tokens that are critical to the description of the event. The definitions of these tokens must not be altered or deleted. If a required token definition is altered, the consumers, filters, and distributors that access the event might stop working. If such a change cannot be avoided, then a new event message should be created and the old event message retired.

EMS Event Message Procedures

EMS provides a specific set of procedures to simplify event message construction and access. The three primary procedures are EMSINIT, EMSADDTOKENS, and EMSGET. EMSINIT builds the header format and requires the generator to mark one token as the subject of the event. EMSADDTOKENS completes the messages by placing the specified tokens into the event message. Event processing applications call EMSGET to extract information from the event message.

Special Event Messages

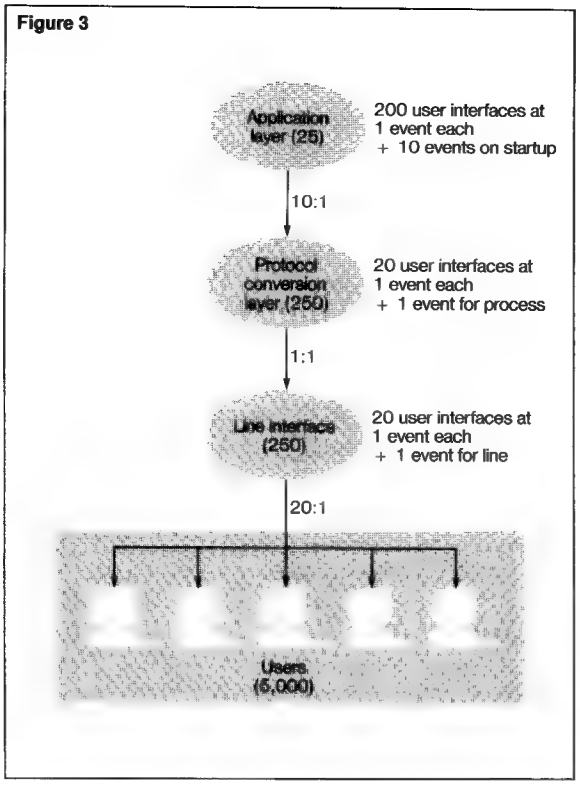
There are two special aspects of event messages defined by EMS: event subjects and action events.

Event Message Subjects. Every event message has one or more tokens designated as the subject of the event. These tokens identify the subsystem object, or objects, most directly involved in the event.

The value of the subject token(s) in the event message is used by the Tandem program VIEWPOINT, the console facility for DSM. Using the subject token values, VIEWPOINT creates a "key." This key is used as an index for that event in the VIEWPOINT event message database. VIEWPOINT operators can see events that occurred on some object (say, a file) independent of which subsystem generated the events since the database is organized by subject, not by subsystem.

Action Events. Action events identify subsystem problems that require human intervention. They are simply event messages that contain a particular EMS-defined token. The action-needed event contains this token with a value of TRUE, while the action-completion contains a FALSE value for that token.

Figure 3.
EMS performance model.



For example, a tape may need to be mounted. Rather than designating a specific terminal or console that must receive this notification, the subsystem generates an action-needed event. This informs an operator (at a terminal of the operator's choice) of the need for the tape mount. Once the tape is mounted, the subsystem generates an action-completion event to EMS. This message notifies all operators that the tape was mounted.

VIEWPOINT supports action events by highlighting action-needed events on a display. This message is dimmed when the corresponding action-completion event is generated.

EMS Collector Performance

Collector performance is critical during system startup and other situtations where events are generated at a high rate. If the collector cannot keep pace with event generation, either event issuers must wait for the collector to accept the events, or events could be lost. This problem is compounded in large system configurations. One of the requirements of EMS was improved performance in a large system environment. Two goals were identified and achieved, meeting the performance requirement. The first was to minimize delays between event generation and event consumption. The second goal was to provide a collector that could tolerate the high event generation rates encountered in large systems.

EMS minimizes the delay between event generation and event consumption by providing event distributors that support only a single event consumer (as compared to pre-DSM architecture). Consumption delays that are created by the rate of processing in the consumer no longer affect the event generators. Event generation rates are limited only by the rate that the collector can accept generated events.

A Performance Model

To meet the second goal, a target collection rate had to be established. A performance model was developed to help determine an appropriate and sustainable collection rate for $0.

The model system supports 5000 user interfaces. Each user interface in the model is a subdevice on a datacomm link that issues one event when the user interface is started or stopped. Each user interface is supported by a protocol conversion layer and an application layer that issues one event when the user interface is started or stopped. User interfaces are collected into groups of 20 for each datacomm line, 20 for each protocol conversion layer process, and 200 for each application layer process. Each line and each protocol conversion layer object issues one event on start or stop. Each application layer process issues 10 events on start or stop. (See Figure 3.)

A system startup sequence was the stress condition used to establish the target event logging rate. The goal was to complete system startup in a 5-minute period. Figure 4 shows the logging rate calculations. Figure 5 shows the calculations used to determine the needed disk capacity. Analysis of the performance results indicated that a collection rate of 52 events per second would be sufficient for this large system configuration.

Since the logging rate to disk is augmented somewhat by the buffer space reserved in the collector, the actual rate that events were written to disk could be slightly less than the 52.5 rate. The difference is calculated by dividing the number of events that the collector can fall behind disk logging by the duration of the high logging rate period.

For startup of a large configuration, the calculation is shown in Figure 6.

This calculation shows that though the actual memory buffer size aids very little in coping with high event generation rates of longer duration, it is extremely helpful in allowing short bursts of high activity. The 52.5 event per second event generation rate for the model configuration during startup can be supported if disk logging can be maintained at 51.93 (52.50 - 0.57) events per second.

Figure 4

5000 subdevices objects @ 1 event each	5000 events
5000 conversion layer objects @ 1 event each	5000 events
5000 application layer objects @ 1 event each	5000 events
250 lines @ 1 event each	250 events
250 conversion layer processes @ 1 event each	250 events
25 application layer processes @ 10 events each	250 events
Total	15,750 events
Total time	5 min

Events per minute = 15750/5 = 3150 events/min
Events per second = 3150/60 = 52.5 events/sec

Figure 5

15,750 events at an average size of 190 bytes	2,992,500 bytes
The default primary extent pages	20 pages
The default pages per secondary extent	100
The number of secondary extents in a file	15
The default total pages for secondary extents	+ 1500 pages
Total extent pages	1520
Page size in bytes	× 2048
Total disk space reserved in a default log file	3,112,960 bytes

Figure 6

Default buffer space available in collector	32768 bytes
Average event size (estimated)	÷ 190 bytes
Events that can be held in available buffers	172 events
Duration of high event generation rate	÷ 300 sec
Average difference between disk logging rate and event generation rate allowed	0.57 event/sec (or 1 event every 1.75 sec)

Figure 4. *Logging rate calculations.*

Figure 5. *Default disk capacity calculations.*

Figure 6. *Startup of a large configuration.*

Disk Logging Rate Performance

To meet the target event logging rate, certain characteristics of the GUARDIAN 90 system had to be considered. In the GUARDIAN operation system, log files can be organized one of two ways: key sequenced or entry sequenced. EMS uses the ENSCRIBE entry-sequenced file organization combined with a positioning function.

Two GUARDIAN disk process options can further affect the achievable rate for writing records to a file. These options, BUFFERED and REFRESH, can significantly affect both performance as well as data recovery.

If BUFFERED = OFF is selected, a physical disk write operation will result from each event sent to the disk process. The alternative setting, BUFFERED = ON, instructs the disk process to accumulate a block of records before writing any messages to disk. The REFRESH = ON setting means that when the first record is written to each new block in the file, the file label is rewritten to disk to update the EOF pointer. With REFRESH set to OFF, the file directory will be updated only when needed or requested.

Figure 7

Default size of each on-line event log file	3112960 bytes
Default number of files retained (= MAXFILE - 1)	×3 files
Default total space available on disk for events	9338880 bytes
Average event size (an estimate)	÷ 190 bytes
Default number of events archived online	49152 events

Figure 7.
For a system where only 100 events are generated per day, 49,152 events represent more than a year's worth of archived messages.

With BUFFERED = OFF and REFRESH = ON, the maximum achievable rate for writing events to a collector log file is about 22 events per second with most common disk drives. (In this model, 4104 disk drives were used.) Disk drive performance is the limiting factor in writing events to the file. This combination, while providing greater data recovery capabilities, degrades logging rate performance.

Tests have shown that with a Tandem TXP™ system using the same disk drives as above, if BUFFERED = ON and REFRESH = OFF, event logging rates of almost 60 events per second can be achieved. With this set of options, disk extent allocation, CPU overhead for generating events, and GUARDIAN 90 message system traffic become the limiting factors in writing events to the file. However, in the event of a double component failure, this combination could affect data recovery.

The EMS default mode sets BUFFERED = ON and REFRESH = ON for log files. The block size for log files is set to 4096 bytes to accommodate the largest possible event (4024 bytes). This combination ensures that the EOF pointer is always current and allows event rates between 40 and 50 per second. If needed, the Subsystem Programmatic Interface (SPI) to the collector allows changes to these option selections when reliability and availability needs so dictate.

Log File Options

Because EMS chains log files, the number of log files used by EMS is theoretically limited only by the amount of available disk space. If no limits on the number of files used are set, EMS could conceivably use up all available space on a disk. To control this, an EMS collector option called MAXFILES allows users to specify how many files to allocate to the EMS log files. The EMS default value is 4.

Most systems will find the default MAXFILE and extent (primary = 20 pages, secondary = 100 pages) values sufficient for retaining a minimal log file archive. For larger systems, the default settings may be changed as appropriate.

The size of the default on-line archive is calculated in Figure 7.

ROTATEFILES. The most common operation problem that can interfere with collector event logging is the failure to delete log files from the log subvolume. The EMS option ROTATEFILES manages the use of the allocated files when the log files reach capacity. The setting (ON or OFF) of ROTATEFILES directs $0 either to recycle disk space or to halt logging.

If the default ROTATEFILES = ON setting is retained, then the oldest log file will be purged, and logging will continue on this newly freed disk space. This recycling of disk space will continue indefinitely. If ROTATEFILES is set to OFF, then logging of events to disk will stop.

If the customer retains the ROTATEFILES = ON default, purged log files cannot be recovered. Event logging does continue, however. Retaining the ROTATEFILES = ON setting assumes that an EMS log file archive is not necessary because events are being forwarded as they occur or log files are being archived regularly.

When Disk Logging Stops. Some customers may choose to set ROTATEFILES = OFF. Failure to remove old log files from the system before the MAXFILE number of log files is filled causes event logging to stop.

When logging stops, a LOGGING STOPPED event generated by EMS informs the customer that old EMS log files from the current log subvolume must be removed before logging can continue. Events are temporarily saved in internal $0 buffers and sent directly to the distributors.

An algorithm in $0 specifies that when a long logging outage occurs, the events just before and after logging stopped and the most current events are saved. Events are sent to the distributors one at a time at the rate they can accept. The distributor filters and processes these events as if they had been read from a log. With this algorithm, if events occur at a rate that the distributors can sustain, as a group, all events that are received will be distributed.

The buffers can store up to 32,768 bytes of information. When less than 4024 bytes (the maximum event size) remains in the buffer after the last event has been sent to all distributors, the newest events in the internal buffers are discarded until 4024 bytes are again available. At this time the LOGGING STOPPED event is once more sent to the distributors.

During the time buffer space is insufficient in $0 for storing a received event, the WRITE, WRITEREAD, or internally defined message used by an event generator to transfer the event to $0 is terminated with an insufficient buffer space error. When buffer space becomes available, events are again accepted and sent to the distributors.

Event Log Processing Performance Optimization

All of the distributors ultimately read event messages from disk log files. Some design steps have been taken to optimize event retrieval performance for the EMS distributors.

Read-Ahead

In order to make events available to a destination as quickly as possible, the distributor, after delivering an event, reads the next event from the log and applies it to the filter. While the destination processes the current event, the distributor continues to read from the log until an event passes the filter. At this point, the distributor must wait until the destination indicates completion or the next GETEVENT command is received.

The Forwarding Distributor collects qualifying events in a block buffer until the target collector completes processing of the last event transmission. The size of the block buffer is 4096 bytes; events are sent unaltered. Performance is improved for all distributors by decoupling reading and filtering of events from the destination's request for the next event or set of events.

Collector-Distributor Protocol

Typically, a distributor processes events in "monitoring" mode; that is, it reports the latest events written to the log. When it encounters an EOF condition, instead of burdening the file system with repeated log read requests, it sends an SPI status command (wait-for-event) to the collector, requesting that a response only be sent if new events have become available. A context is submitted for reference, representing the distributor's current position (log file name and record address) in the log. After a reply has been received from the collector, the distributor continues to read the log. The status return contains a new context that the distributor must examine in order to detect a log switch. If the collector has started a new log, the distributor must still continue to read the current log until an EOF is encountered again. This is because events may have been written to the log since the status request was sent. After the EOF is detected again, the distributor switches to the new log file indicated in the status return.

All status requests are sent no-waited. The distributor can thus service multiple I/O requests. When events are received from more than one source collector, they are ordered by log time and merged onto a queue. The queue holds no more than one event per collector. After the oldest event on the queue has been processed, the distributor issues another read request to the log from which the last event had been received. It is possible that no event is available from any of the source collectors; in that case, "wait-for-event" status requests are pending for each source.

The distributor-collector protocol uses an internal SPI programmatic interface. This protocol reduces CPU time by eliminating unnecessary calls to the file system. When few events are being generated, the overhead for each event involves two disk reads (to get the event and to get the EOF), and a collector status command. When many events are being generated, the overhead for each event involves only one disk read (to get the event).

Event Selection

EMS provides tools that allow an application or an operator to selectively retrieve events for processing. These tools are positioning and filtering.

Positioning allows a user to select events that start at a specified date and time. A position command can be given to a distributor via programmatic interface or as a startup parameter. The position may be selected anywhere within the chain of logs currently associated with the source collector(s).

Filtering allows a user to select events according to a logical evaluation of tokens and token values. Filters may be installed at startup or programmatically.

Positioning

Since event logs are entry-sequenced files, search keys for quick access are not readily available and are too expensive (in terms of system resources) to build. A sequential search of one or more potentially large log files is too time consuming and therefore not feasible. The solution is a binary search algorithm that yields a block of events which are then scanned sequentially.

Binary Search. A binary search is conducted over (at most) one log file. The search begins with the log currently associated with the source collector. Assuming a chain of files, the target log is determined by following the linkages provided in the first event of each log. If the timestamp of the first event in this log is greater than the positioning time, the next log is selected, and so on. If the end of the chain is encountered and the timestamp in the first event is still greater than the positioning time, a warning message is generated, and the current position set to the first record in the last log inspected.

Assuming a minimum event size of 108 bytes, a 4096-byte block contains up to a maximum of 37 events. With 10 log files of 3.2 Mbytes each, a worst case binary search would take approximately (log2 (900) + 10) = 20 disk accesses. At 50 ms per disk access, the search time would be approximately 1 second.

When more than one source collector is configured, the logs for each collector are processed in parallel. This is done by using a distributor's reentrant I/O processing loop. If the logs are configured on different disk packs, performance will be optimized further.

Event Log Chains. After a positioning command has been issued, the distributor is said to be in "historical" mode. It begins to read events starting at the new position and continues until it catches up with the collector's current position. The names of the log files skipped during the positioning search are recorded in a name table created by the distributor. This name table is referenced during event retrieval. If the table cannot hold all the log file names in the chain (maximum of 10), a new table will be built once the most recent log from the list has been processed. The new names are obtained by reading the first event in each log, starting with the collector's current log, and continuing in reverse order until the most recent log is processed.

If a broken link is encountered during event processing in historical mode (a log file obtained from the name table cannot be accessed or has an invalid link in its first event), the distributor generates an informational event and attempts to recover by skipping to the next available log file, as determined from the name table. If a next log is not available, the distributor stops the search and an appropriate event message is generated. Event processing continues if other source collectors are still functioning.

Log Time vs. Generation Time. Events are always stored in log time sequence; that is, the timestamp is provided by the collector at the time the event is written to the log. Events are not necessarily in generation time order, especially if skewing has occurred after event forwarding from several sources. An event consumer is normally more interested in the event's generation time. These two times can differ considerably.

After determining the target log file, the distributor conducts the binary search for a target block that contains a record whose log time is greater than or equal to the positioning time specified. After completing the binary search, and assuming that positioning by generation time is selected, the distributor must now search forward within or beyond the target block until the generation time is also greater than or equal to the positioning time. From that point on, all events with a generation time greater than the positioning time qualify for distribution.

The presence of irregular events in the log must also be considered. Irregular events are those events that have a log timestamp smaller than 1JAN1974, indicating that a cold load occurred. Irregular events must be skipped during the binary search and during the block search. After the positioning completes, any irregular events encountered will be processed.

Filtering

Filtering is a programmatic way of selecting specific events for processing. A filter is installed in the appropriate distributor and only those events that meet the filter's specifications are passed to the consumer.

Filtering is used for different purposes: it is used by event processing applications to select specific event messages, to select the set of events a Forwarding Distributor sends to a remote node, and to select the set of events that the Printing Distributor displays on its output device.

To meet these diverse needs, EMS provides a filter language to express selection criteria. A filter compiler, EMF, accepts the filter language as input and produces a filter object file. The filter object is loaded into an EMS distributor, which selects events for distribution based on the filter's specifications.

The filter language allows the user to test for the presence or absence of an arbitrary token in the event messages as well as testing the value of any token. Like other languages, these tests can be combined using AND, OR, and NOT. Filter execution is controlled by an IF statement.

For example, a simple filter that will pass only C00 PATHWAY™ transaction processing system event messages might be written:

```
Filter pass^pathway;
Begin
  if ZSPI^TKN^SSID =
  TANDEM.PATHWAY.C00
  then pass;
End;
```

Changing Filters
Filters can be changed by simply using the application to load a new filter into a distributor. Though simple, the disadvantage to this approach is that all filters the application might want would have to be known and all such filters compiled before the program was run. A better approach uses parameterized filters.

Parameterized Filters. Filter parameters may be changed by the application program after the filter has been loaded into the distributor. The parameters simply take the form of token values. Parameters are named, or distinguished from one another, by their token codes, which are defined when the filter is written. The filter source language provides the same operations on parameter tokens as for the event message tokens. In essence, this allows filter execution to be controlled via the parameter tokens loaded by the event processing application rather than preset at filter compile time.

Installing Filters
With the programmatic interface, filters can be installed and changed while the distributor is running. Combined with the positioning command, this is a powerful tool for problem tracking, particularly when filter parameters are introduced. Automated applications can be written that select filter parameters according to the outcome of a previous log scan, for instance.

The distributor ensures that events are not skipped after a filter has been changed. Because of the read-ahead feature, the distributor must reposition in the log so that all events that were received before the filter had been changed, but not seen by the consumer, are applied to the new filter. This makes filter changes transparent to the consumer.

Distributing Tokenized Messages as Text

In EMS, event generation and text formatting are distinct operations. In certain cases, a tokenized event message must be formatted into text. This is true for $Z0, the Compatibility Distributor, and in situations where messages are displayed or printed. Formatting functions are provided by EMSTEXT, a callable procedure in the GUARDIAN 90 system library. Any process can access the formatter by simply "calling" EMSTEXT.

EMSTEXT allows all reasonable data types used in tokens to be converted to text. EMSTEXT converts the tokens into text with the use of a formatter template. This template, provided by the event designer, supplies the initial definition of the text.

EMSTEXT expands the template to text using the data from an event passed by the EMSTEXT caller. Templates for all events are provided by the event designers and may be modified by the user to meet specific needs. Because the text definitions exist in the templates, customers can tailor the text display of an event to suit their own environment. This is desirable in non-English speaking situations. Considering that another language may be more common than English at a site, event messages displayed in the native language would be easier to understand.

Text Tokens. In situations where text templates cannot be used, EMS defines a text token. A text token represents a line of text. EMSTEXT decodes this, adds the appropriate header, and returns it as a text message. Using text tokens, users can report and display management information without providing text templates for events. However, text tokens should be used sparingly. The increased space needed to save the text in each reported event and the inflexibility inherent in text being part of the event contents make excessive use of text tokens unwise.

Compatibility Considerations

It is important to ensure that the correct text is always displayed even if the event is generated on one system and the text is formatted and displayed on a system with a different GUARDIAN 90 version installed. A text display must also be generated for subsystems that report pre-DSM management data.

Version compatibility for event services applies only to GUARDIAN releases after C00 (which support EMS). To ensure version compatibility, guidelines for event and template modification allow changes to the text template that clarify its meaning or correct mistakes. The text generated by EMSTEXT for an event should never be used by management applications to programmatically analyze event contents. The text version of events is meant for human consumption. If events need to be analyzed by a program, the tokenized version of the message should be used.

Conclusion

EMS provides a centralized event collection facility that supports reporting management data with old operator messages as well as with new tokenized events. Supporting the old and new messages allows for orderly migration to EMS by existing subsystems.

Subsystems within the GUARDIAN 90 environment can define and generate tokenized event messages to meet all management data needs. Each event message describes a noteworthy occurrence in the network in a timely manner. The techniques used to tokenize management data provide both required and optional information in each event. EMS provides system procedures to aid customers in creating events from management data. Interrelease changes to management data are anticipated by providing guidelines for the creation and extension of event contents.

EMS uses disk log files to queue and save reported events. Separating event collection from event distribution improves the rate of event collection. Mechanisms built into the collector and distributors ensure that management data can be reported when needed, even if a system component fails.

By allowing the user to configure many distributors where each supports a specific user with a specific type of service, EMS facilitates the use of a variety of management applications. The three EMS distributor types retrieve and use filters to select event messages from collectors or disk log files. EMSTEXT, a procedure which can be accessed by any process in the GUARDIAN 90 system, produces a text display from an event.

Hank Jordan is a software designer in DSM software development. His experience prior to this assignment primarily involved designing data communications device interfaces.

Randy McRee received his B.S. degree in E.E.C.S. from the University of California at Berkeley in 1981. Randy joined Tandem in 1984 as a network analyst for Tandem's Network Support Group. In February of 1985, he joined the DSM group in Software Development to work on the EMS project.

Rudy Schuet is a software developer with an M.S. in Electrical Engineering from the University of Karlsruhe, West Germany. Since joining Tandem in 1982, Rudy has contributed in the areas of transaction monitoring (TMF) and, most recently, in network management, where he designed and implemented the event distributor component. Prior to joining Tandem, Rudy was the key developer and project leader for a relational database management system.

Data Replication in Tandem's Distributed Name Service

Distributed Name Service (DNS™) is part of the set of Tandem's system management products known collectively as Distributed Systems Management (DSM). DNS simplifies the management of objects in a NonStop™ system or an EXPAND™ network by managing a distributed, and partially replicated, database of names of those objects.

This article provides practical solutions to many of the problems associated with data replication. It covers four general areas:

- The services provided by DNS.
- Network considerations, including the characteristics of name management that require the use of data replication.
- The replication mechanism implemented in DNS.
- DNS architecture.

Services Provided by DNS

There are many named objects within an EXPAND network. GUARDIAN™ file names are really addresses since they are chosen by system administrators and are not independent of network location.[1] Subsystem-defined names, such as PATHWAY™ names, while "readable by humans and of mnemonic value" (Terry, 1985) are only usable within the context of a given PATHWAY application.

The purpose of DNS is to allows users to assign names, or *aliases*, to subsystem-controlled objects (SUBSYSTEM OBJECTS). DNS allows definition of an arbitrary number of aliases for each object.

[1]Douglas Terry makes the distinction that names are chosen by users, whereas addresses are assigned by the system or system administrators. Names are characterized as being readable by human beings, of mnemonic value, and independent of network locations.

DNS also allows definition of a single name (COMPOSITE) that refers to several related subsystem objects. Frequently, a single object is known to multiple Tandem and/or user subsystems; e.g., an automated teller may be known to PATHWAY, SNAX/HLS, and SNAX. Through use of DNS composites, such objects can be assigned a single name.

Users can define arbitrary *groups* of aliases and composites. Groups may themselves be members of other groups.

The principal client of DNS is expected to be network management applications (NMA). Among the functions provided to NMAs by DNS are:

- Translating an alias to a subsystem object name (e.g., when turning a command specifying an alias or composite into the appropriate subsystem command with the correct subsystem object name). This allows construction of command interpreters that accept meaningful names.
- Translating subsystem-object names (e.g., in events) to aliases. This allows event processing applications to report meaningful names to operators. Facilities are included for distinguishing among the various aliases for an object, thus allowing the application to select for display the alias most appropriate for the audience.
- Translating a group name to the names of the members of the group, allowing NMAs to implement group-oriented commands; for example, SF-ATMS includes ATM01, ATM37, ATM44, etc.

Network Considerations

While DNS can be used in any operational environment, it is primarily designed to facilitate centralized management of a distributed network of systems.

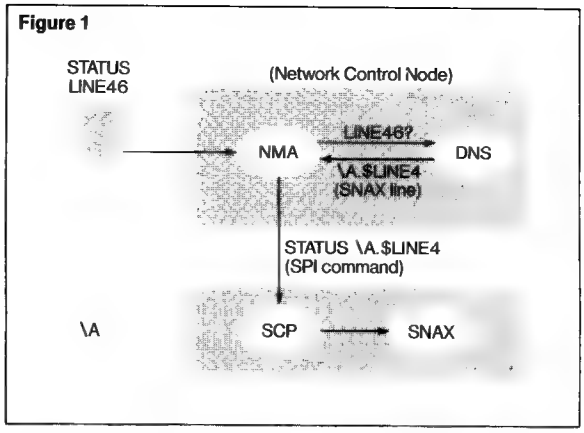
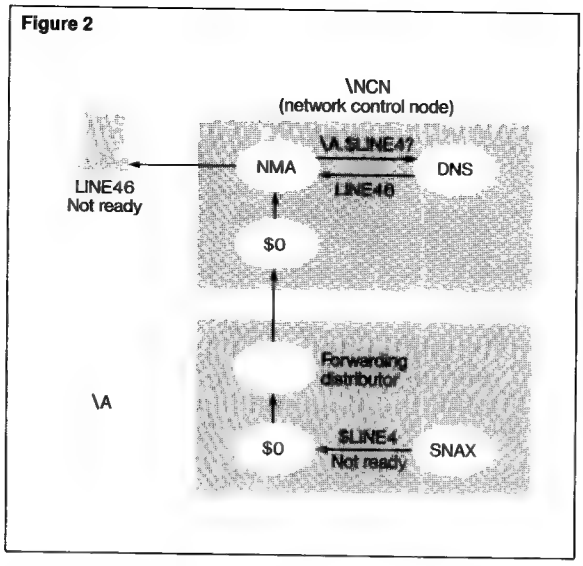


Figure 1.
Alias-to-object-name translation.

The DNS database on a network control node (NCN[2]) needs to know about objects on nodes controlled from that location. This allows NMAs running at NCNs to use a single set of user names for objects at multiple nodes when communicating with the human operator, and simultaneously to communicate with subsystems using the appropriate subsystem-object names for the objects. (See Figure 1.)

[2]For the purposes of this article, the term "network control node" refers to a system (the control node) where human operators control the operation of all or part of a network of Tandem systems.

Figure 2.
Object-name-to-alias translation.



Similarly, NMAs that process events are able to report problems to human operators at network control nodes using aliases. (See Figure 2.)

Where network control functions are centralized, disaster recovery plans must provide for transfer of control from one NCN to another.

From the point of view of DNS, this means that for each name known to an NCN, there must be another NCN in the network that also knows about that name.

Additionally, it is possible for any system to become isolated from the rest of the network; therefore, it should be possible to control each system either locally or via a dial-up terminal. This means that DNS name translation should be available on every system for at least those objects residing on that system.

Replication of Name Definitions

Although it is desirable that the definition of names be stored at multiple systems in an EXPAND network, it is unreasonable to expect DNS users to define each name to each system where that name might be used. Rather, each name is defined on one node, and DNS replicates that definition on other nodes. The set of nodes that contains a name's definition is referred to as the name's *domain* and is chosen by the user.

Time-Staged Replication

The simplest way of performing this replication is to use a "write all, read any" scheme. Under such a strategy, a single transaction is used to add a new name definition or change an existing name definition; all copies of the definition are changed or none of them are changed. This ensures that all copies of a name's definition are consistent, but it severely limits update availability; if one of the copies of the definition were unavailable, the name could not be changed. More specifically, if one of the systems in a new name's domain were unavailable, the name could not be added.

The limitations of write all, read any replication led to investigation of other means of data replication. A time-staged strategy was finally adopted.

When a new name is added, its definition is initially stored on only a single system; DNS refers to this system as the name's *definition node*. The name's definition is subsequently "exported" to each of the systems in the name's domain; i.e., the name's definition is added to the DNS database on each system in the user-specified domain. In this way, if one of these systems is temporarily unavailable, the name may still be exported to the other systems in the name's domain. When the previously unavailable system is once again accessible, the name's definition is exported to that system as well.

Changes to the definition of existing names are processed in the same way.

Because changes in a name's definition are exported asynchronously to the systems in the name's domain, it is possible for the various copies of that definition to be temporarily inconsistent.

Once a change to a name's definition is exported to all the systems in the name's domain, all copies of that definition are consistent.

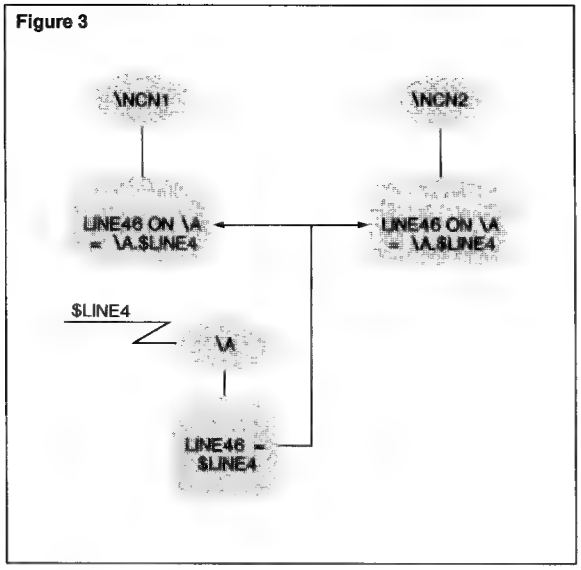
Avoiding Duplication and Incompatibility

When a new name is defined at a node, the other systems in the name's domain are not consulted. Consequently, there is a possibility that the new name is a duplicate of an existing name defined on another node in the new name's domain. To avoid such name collisions, each DNS name is qualified by the name of its definition node. For example, if the name LINE46 is defined on system \NEWYORK, the fully qualified form of that name is LINE46 ON \NEWYORK. This fully qualified form need only be used, however, on nodes where more than one LINE46 is known (e.g., LINE46 ON \NEWYORK, LINE46 ON \CHICAGO).

If users at any node in a name's domain were free to update the definition of a name, it would be possible for users at two different systems to make simultaneous incompatible changes to a name's definition. DNS avoids this problem by only allowing a name's definition to be changed at the system where the name was originally defined (i.e, the name's definition node).

Situations might arise where a name's definition node is unavailable and a change to that name's definition is required. DNS permits users to make a modifiable copy of remotely defined name definition; any changes made to such a copy are strictly local and are not exported to the other systems in the name's domain.

Figure 3.
Name replication.



DNS databases are protected by the Transaction Monitoring Facility (TMF). When a name definition is exported, a TMF transaction is started by DNS at the exporting system. This transaction is used to update the DNS database at the importing (remote) system; if the export succeeds, DNS at the exporting system updates its own database to indicate that the name definition has been successfully exported and the transaction is committed. If any errors occur in this process, the transaction is aborted, thus restoring both the local and remote databases to their prior state.

The partial replication scheme employed within DNS results in DNS databases being node-specific; in other words, a DNS database created on one system cannot be moved to another system using standard utilities like FUP or BACKUP/RESTORE. Equivalently, the node name and node number of a system should not be changed once DNS is installed.

As shown in Figure 3, the recommended way to use DNS replication is to define each name on the system where the underlying subsystem object(s) reside. For example, if LINE46 is to be an alias for \A.$LINE4:

1. $LINE4 is defined to DNS at \A.
2. LINE46 is defined at \A as an alias for $LINE4.
3. The domain of LINE46 should consist of one or more network control nodes (\NCN1 and \NCN2 in Figure 3).

By defining LINE46 in this manner, it is possible to change the master copy of the definition (i.e, the copy at the definition node) of LINE46 any time that the underlying subsystem object (\A.$LINE4) is accessible. If LINE46 were to be defined at \NCN1 and that control node became unavailable, it would not be possible to change the master copy from \NCN2 even though \A.$LINE4 was available from \NCN2.

DNS Architecture

DNS consists of three program components: DNSCOM, the name manager, and the name exporter. The relationship between them and the database is shown in Figure 4.

The name manager and name exporter communicate via the GUARDIAN file system with the name exporter being the requester and the manager being the server. In addition to this IPC (Interprocess Communication) interface, the two processes communicate through use of a queue file.

The process of exporting name definitions involves the local name manager and name exporter, as well as the name managers on one or more remote systems.

DNSCOM

DNSCOM provides an interactive interface to DNS. It is used to:

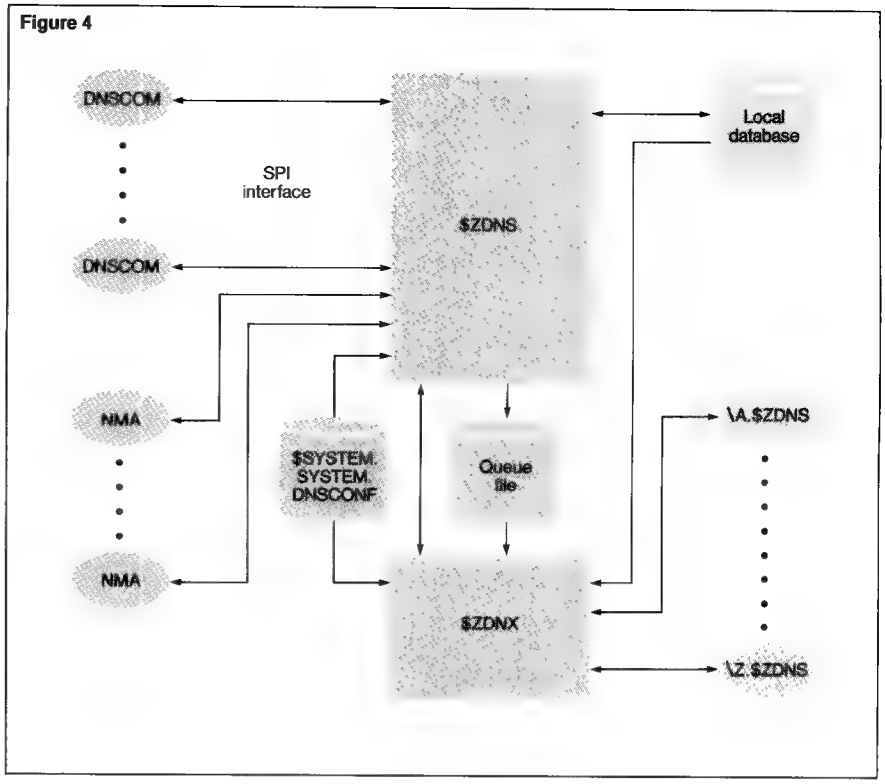
- Initially create the DNS database and the $SYSTEM.SYSTEM.DNSCONF file.
- Control the various DNS processes.
- Perform inquiry and update operations against the DNS database.

Name Manager

Each node on which DNS is installed has a DNS name manager called $ZDNS. The name manager is a multi-threaded NonStop process which accepts SPI (Subsystem Programmatic Interface) requests from NMAs and performs I/O operations against the local DNS database.

The name manager is a TMF server; that is, it expects to acquire TMF transactions via $RECEIVE. Consequently, the NMA (or DNSCOM) has control over TMF transactions.

Figure 4



Name Exporter

Each network node where DNS is installed has a name exporter that is responsible for all export operations. The name exporter is a multi-threaded process that runs with the reserved name $ZDNX.

Figure 4.

DNS architecture. DNS maintains its configuration in $SYSTEM.SYSTEM.DNSCONF. This file is created during processing of the INITIALIZE DNS command and serves as a safe-store for the values of the various DNS configuration parameters.

Figure 5

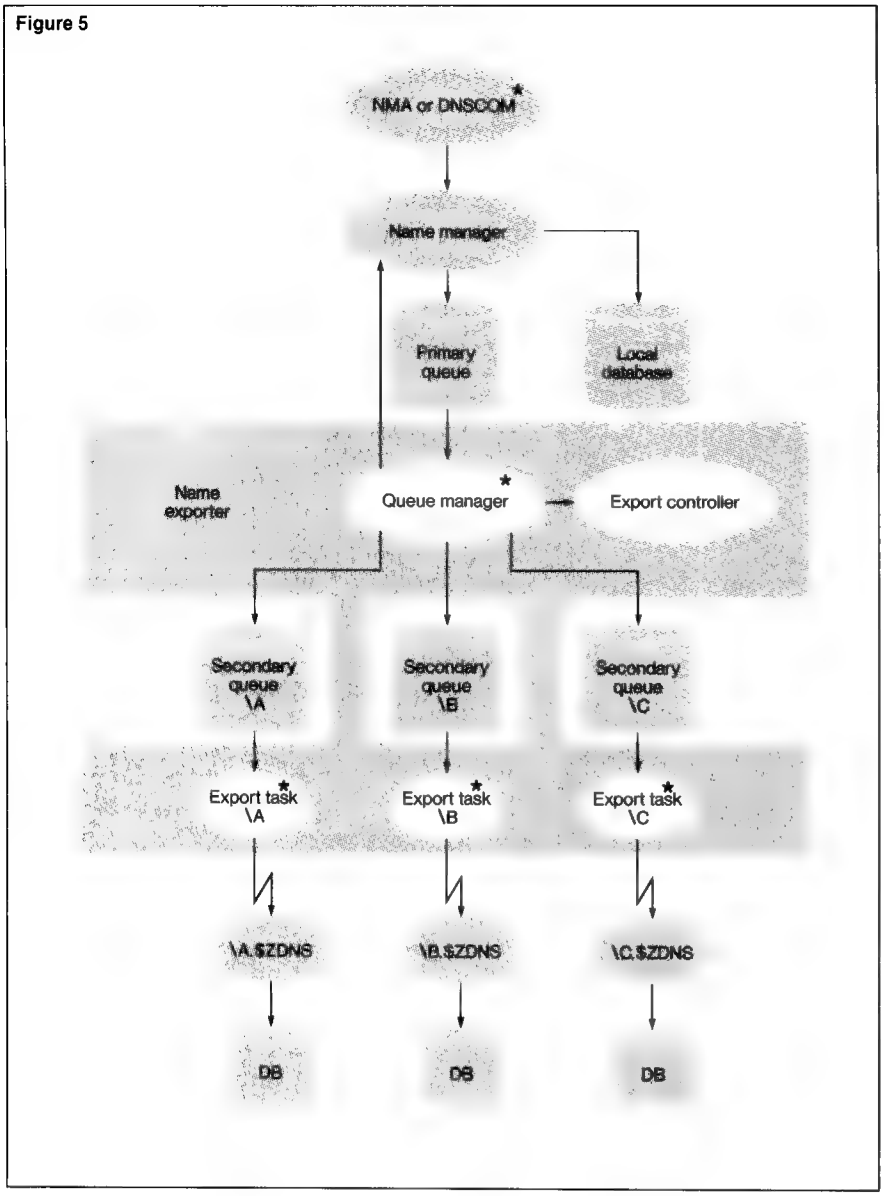


Figure 5.
Exporter architecture. Points at which a TMF transaction occurs are flagged with an asterisk (*). The number of TMF transactions required to add a name and replicate the name's definition on N nodes is 2 + N.

Placing the export function in a process separate from the name manager considerably simplifies the overall design. As previously described, name export involves the use of TMF, which means that the process that initiates name export must be a TMF requester (must call BEGINTRANSACTION). Coding and testing multi-threaded TMF requesters is greatly simplified if the process involved does not need to run as a NonStop process pair. Consequently, the exporter does not run as a NonStop process pair; if it fails, it is automatically recreated by the name manager.

The Queue File

The queue file can actually be thought of as three separate logical files.

Primary Queue. The primary queue serves as a place for the name manager to store export requests until they can be acted upon by the name exporter. In addition to providing a safe store for these unprocessed export requests, the primary queue acts as a buffering mechanism between the name manager and name exporter. This buffering allows export requests to be generated by the name manager at a rate that is independent of the rate at which the name exporter is able to process them.

Secondary Queues. The queue file can contain one secondary queue for each remote system in the EXPAND network. Each secondary queue holds export requests destined for a single remote system.

Export Control File. The export control file portion of the queue file is used to store control information.

The Export Process

As the name manager processes an ADD, ALTER, or DELETE command involving a replicated name definition, it inserts export requests into the primary queue. Since the queue file is a TMF-audited file, the write to the primary queue is done under the same TMF transaction as the database update; if that transaction is subsequently aborted, the export request in the primary will be deleted during TMF transaction backout. The export process is illustrated in Figure 5.

The name manager makes no attempt to determine which systems are to be involved in the update. That task is delegated to the part of the name exporter known as the *queue manager*. The queue manager's main processing loop is as follows:

1. Begin a TMF transaction.
2. Read the primary queue with lock.
3. If a request is found, analyze the request to determine the systems to which it needs to be sent. For each of these systems, insert a copy of the request into the corresponding secondary queue and delete the primary queue entry.
4. End the TMF transaction.
5. If the primary queue is empty, issue a command to the name manager whose completion will signal when a new request has been inserted into the primary queue.

Note that the name exporter issues a request to the name manager only when the primary queue is empty; as long as there are requests in the primary queue, the exporter processes them without involving the name manager. The buffering provided by the primary queue allows the name manager to process update requests without having to wait for the name exporter.

In addition to the queue manager, the name exporter contains an *export controller* and up to 31 *export tasks*. The purpose of the export controller is to monitor the status of secondary queues and to initiate export tasks when required and to monitor export task execution. When initiated, each export task processes the secondary queue for a single remote system. The export task's main processing loop is as follows:

1. Begin a TMF transaction.
2. Read the secondary queue with lock.
3. If end-of-file, then stop.
4. Otherwise, format the SPI request and send it to the remote manager.
5. If there is no error, delete secondary queue entry and end the TMF transaction
6. Otherwise, abort the TMF transaction and stop.

Conclusion

DNS allows users to assign meaningful names to both individual objects and sets of objects. To be useful in a network operations environment, DNS must be able to translate between these names at multiple systems within the network. This requirement dictates that portions of the DNS database must be replicated within the network.

In a large network, the chances of some node being off-line are higher. This means that, at any given time, an attempt to update a large number of copies of a name's definition is more likely to be unsuccessful.

By employing a time-staged strategy for database replication, DNS is able to maximize update availability while at the same time guaranteeing that the multiple copies of a name's definition will ultimately converge.

Reference

Terry, D.B. 1985. Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments. PROGRESS Report No 85.4. Computer Science Division (EECS). University of California, Berkeley.

Tom Eastep is a software developer for DSM products and is the designer of DNS. Since joining Tandem in 1980, he has also been a systems analyst and analyst manager.

SCP and SCF: A General-Purpose Implementation of the Subsystem Programmatic Interface

Tandem has developed two new data communications network management products, the Subsystem Control Point (SCP) and the Subsystem Control Facility (SCF), that use the Subsystem Programmatic Interface (SPI) as their base. By using SPI procedure calls and their own layer of implementation standards, SCP and SCF are interfaces that are easy to use, consistent, and low maintenance. The development of SCP and SCF as network management interfaces for such diverse subsystems as EXPAND™, SNAX, and X25 demonstrates the versatility of SPI and its value in the development of new interfaces and network management applications (NMAs).

This article discusses some of the design issues encountered during the development of SCP and SCF. The programmatic interface to SCP is discussed first, followed by the human interface to SCF. A general understanding of SPI is required to understand this article.

The Programmatic Interface

The data communications programmatic interface is the method by which NMA requesters talk with the data communications subsystem servers. SPI provides a set of procedure calls and programming standards for building and dismantling network management requests and responses. The data communications programmatic interface is actually a diverse set of separate programmatic interfaces. Uniting these interfaces was the main goal of the SCP design team. This section describes issues, such as consistency, simplicity, and reliability, that facilitate NMA requester design. (See Figure 1.)

Consistency Across Subsystems

The first concern was to present a consistent set of programmatic interfaces across all data communications subsystems. To provide this level of consistency, data communications programmatic interface standards were developed in addition to the SPI standards. These standards include definitions of common error messages, command syntax, command semantics, and constant values.

Developers of NMAs receive many benefits from these extra standards, including reduced memory requirements, simplified coding, and quicker familiarity with new subsystems. (Refer to the *Communications Management Programming Manual* for documentation on these standards.)

Application Development Simplicity

Because an NMA must be able to control more than a single subsystem per system, a mechanism was needed to reduce the number of opens an NMA must maintain. Without reducing the number of opens, the NMA would have to be extremely complex. For example, Figure 2 shows the number of process-to-process opens that an NMA must maintain in a network with four systems and four subsystems per system.

The solution was to introduce one SCP process per system through which all data communications management traffic would be channeled. Figure 3 shows the same configuration as Figure 2 with the addition of the control point process. NMA complexity is greatly reduced by using a control point process where there is a one-to-one relationship between the number of systems and the number of opens an NMA must maintain.

Interface Reliability

With all of the data communications subsystems controlled through a single process, the next concern was reliability. If SCP were to terminate or become overwhelmed with requests, all data communications subsystems would be inaccessible.

The issue of SCP terminating was addressed by requiring SCP to run as a NonStop™ process pair. The issue of being overwhelmed with requests was addressed by allowing multiple copies of SCP. Because SCP utilization is implementation-specific, it was decided to provide several SCP configuration options. The default configuration is a single SCP process pair named $ZNET. This should fulfill the needs of the majority of Tandem customers. SCP can also run as a set of context-free server processes (as in a PATHWAY™ server class). This approach is more sophisticated and is applicable to customers who wish to write NMAs based on PATHWAY or expect a high volume of SCP traffic. The last SCP configuration consists of a single SCP process per NMA. Usually this is used in cold start situations where throughput is critical.

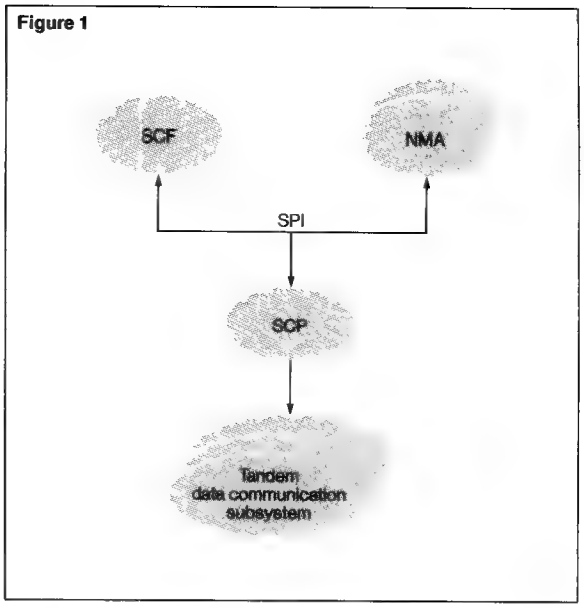


Figure 1.
The relationship between SCF, NMAs, and SCP to control Tandem data communication subsystems, using SPI.

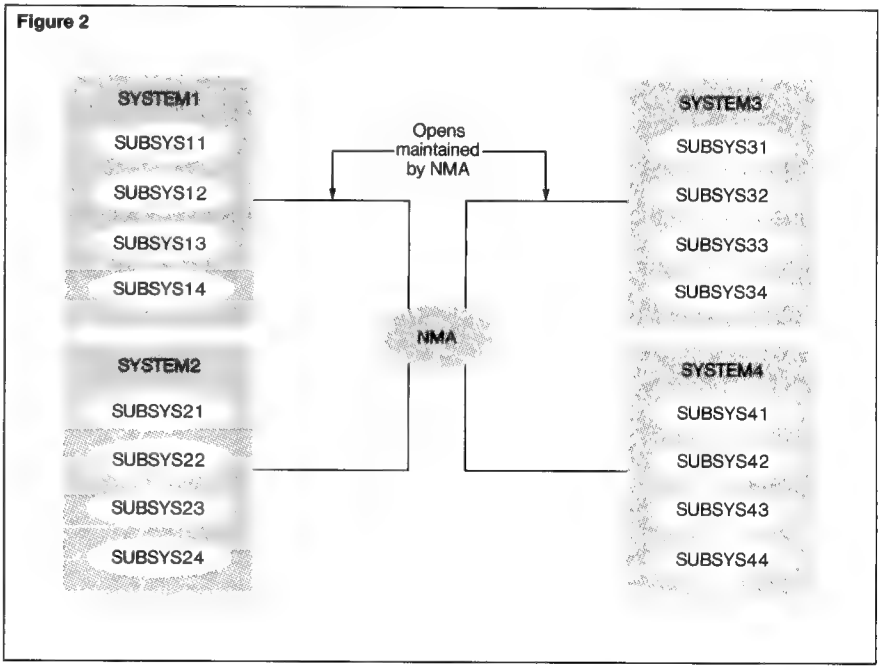


Figure 2.
Process-to-process opens without SCP.

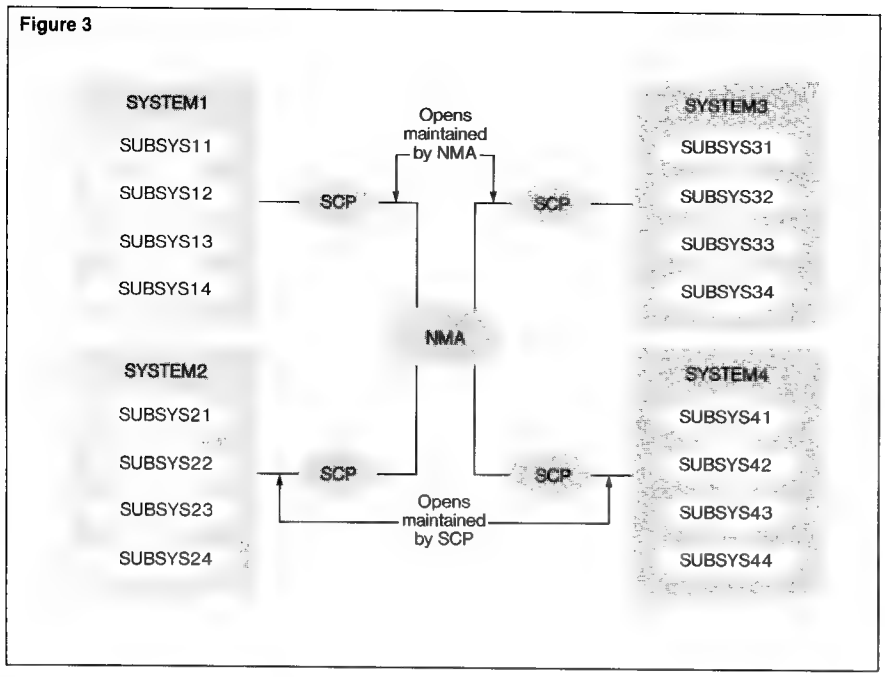


Figure 3.
Process-to-process opens with SCP.

Provided Common Functionality

An additional benefit of a control point process, such as SCP, is that it can provide some common functions to data communications subsystems. This allows for greater code leverage and reliability. Currently, the common functions provided by SCP are trace initiation, version checking, and a basic level of security.

Trace initiation for an I/O process involves the allocation of an extended segment and the linking of that segment to a disk file name. This may also include starting a trace collector, which collects large amounts of trace data.

Version checking guarantees compatibility between NMAs and subsystems. SCP compares the version level of NMAs with the version level of subsystems (on a per command basis) and returns an appropriate error response when an incompatibility is detected. This scheme protects the user without being overly restrictive.

SCP also provides security against unauthorized users changing the configuration of the system. Those commands that can change the state or configuration of a subsystem must be issued by a "super group" user or a user in the same group as the initiator of the subsystem. All other commands are considered nonsensitive and can be issued by any user. If SCP encounters a sensitive command issued by an unauthorized user, the request is rejected with an appropriate error response.

The Human Interface

The human interface allows the user to control the subsystem by entering text from a terminal or an OBEY file. While designing SCF, the developers considered many issues that would make the human interface maintainable, reliable, and user friendly.

Maintainability of the Human Interface

The number of subsystems supported by SCF was expected to be large (greater than 30), and subsystem functional needs were expected to increase with time. Experience with existing data communications command interpreters shows that a state of continual change in the human interface is the norm. This high volume of change, at times, has jeopardized the timely delivery of new features or products.

To make SCF easily maintainable, the developers borrowed the concept of product modules from a similar Tandem product, PTRACE. As with PTRACE, SCF is made up of a set of kernel procedures and a set of product modules. Product modules are blocks of subsystem-specific code compiled independently of the kernel portion of the program and then linked to it at a later time. This type of product module architecture allows for parallel development of new features without the possibility of one subsystem's changes affecting another's.

Product modules use a common set of library procedures to perform functions such as command line parsing, SPI buffer maintenance, communication with the subsystem, error message display, data conversion, and information display. Only subsystem-specific aspects of commands need to be implemented.

The kernel consists of a core set of procedures that act as a buffer to the product modules, invoking them only when necessary. The kernel provides environmental commands, such as ASSUME, ENV, EXIT, FC, HISTORY, LOG, OBEY, OUT, RUN, SETPROMPT, SYSTEM, TIMEOUT, and VOLUME. It also provides common functionality, such as abbreviations, aliasing, break key handling, command history, error recovery, and I/O file management. This architecture satisfies the needs of evolving subsystems at a minimum cost without sacrificing functionality.

Consistency Across Subsystems

With so many product module developers contributing to SCF, there was concern that implementation inconsistencies might occur. The programmatic interface standards leave considerable room for unique human interface interpretation.

The consistency concern was addressed, in part, by having the kernel parse the first two tokens (command and object) that are common to all product modules. Other consistency issues were resolved by establishing standards for the command syntax and display formats.

Compatibility with Existing Products

One of the SCF design criteria was compatibility with existing data communications command interpreter syntaxes so that users would not have to learn a new human interface and, at the same time, incorporate the time-saving features of new command interpreters, such as TACL (Tandem Advanced Command Language).

The syntax of the predominant data communications command interpreter (CMI) was used as the basis for SCF's syntax, with very few exceptions. As a result, existing CMI users can learn SCF and convert CMI OBEY files to be used by SCF with very little effort.

The requests for added functionality were answered by incorporating a number of TACL features and commands into SCF. These features include aliases, function keys, a tailorable prompt, a command history buffer, a help key, and a custom file. They provide an easy transition between TACL and SCF and, again, reduce the amount of learning required to use SCF. By combining the syntaxes of old and new products, customers' learning investments were protected, while providing a higher level of capability.

Figure 4.
An example of a detailed SCF error message. SCF's prompt is an arrow (→).

Figure 4

```
→TRACE PROCESS $ZNET, TO TRCDATA1

SCP #-00822 Trace is currently active for $ZNET

Probable Cause

A TRACE command has been received. The
TRACE facility is currently active in the target
subsystem, and an attempt was made to start
another trace.

Recommended Action

Stop the current trace. Then, reissue the request.
```

Figure 5.
An example of SCF's confirmation messages.

Figure 5

```
→ASSUME LINE $X25BIT
Assume..... LINE $X25BIT
→START
START accepted by X25AM: LINE $X25BIT
```

Figure 6.
An example of the ALIAS command.

Figure 6

```
→ALIAS F FUP FILES
→F
$DATA.STEVE
ARTICLE ARTICLE1 BLOCK BRUCE CLOCK
DOC1
```

User-Friendly Features

The expertise of the SCF users was expected to range from expert to novice. In order to facilitate the needs of both types of users, SCF supports two levels of operational expertise for the HELP command, error messages, and entering commands.

In HELP command line mode, the user enters the full help command on a single line. This mode of operation is fast but requires some prior knowledge of SCF syntax to properly enter the command.

The HELP command menu mode of operation is entered by typing the command, HELP. The user is then prompted with the appropriate options, and stays in the menu mode as long as desired. This option is a slower way of getting information but requires little knowledge of SCF syntax. The menu mode feature cannot be invoked from OBEY or custom files. An SCF custom file is a personal OBEY file containing commands to be executed at the beginning of each SCF session.

Error messages from SCF are generally only one line long, which may not be detailed enough for novice users. For additional information, the user can consult the SCF manual, which includes a detailed explanation and recommended action for each error message; or by issuing the command, DETAIL ERROR ON, additional information will be displayed on the screen following all error messages. Figure 4 gives an example of a detailed error message.

Errors and warnings are always displayed, but by default, SCF does not report the results of successful non-informational commands (e.g., START, ASSUME). For experienced users this may be adequate, but for inexperienced users the results of some commands may not be so obvious. SCF can be configured to display command results for both error and normal cases. This option is enabled with the command, CONFIRM ON. Figure 5 gives an example of a confirmation message.

To save the user from repetitive typing, SCF has a feature called *aliasing*. Aliases are commands or parts of commands that can be represented by a single keyword. Aliases are defined by the ALIAS command and can appear either at the beginning of a command line or anywhere in the command line. If they do not appear at the beginning of a command, they must be preceded by a hyphen (-). The alias name is expanded to the value of the alias prior to command execution. Recursive definitions of aliases are not allowed. Function keys are implemented as aliases F1 through F15 and SF1 through SF15 (F16 and SF16 are reserved for the help key and exiting SCF, respectively). Figure 6 gives an example of the ALIAS command.

Finally, SCF allows key words to be abbreviated to the least number of characters needed for recognition. Any unique abbreviation of a word is acceptable (e.g., SETP and SETPRO are both valid abbreviations for the command SETPROMPT). This implementation is the same as the command abbreviations allowed in the text editor program PS TEXT EDIT™ (TEDIT). The only difference is that SCF allows abbreviations for all key words, not just commands. Key words contained in OBEY files and alias names cannot be abbreviated. The alias name restriction is to avoid the confusion of a dynamically changing set of valid abbreviations, while the OBEY file restriction is to ensure OBEY file compatibility in the event that new key words are added in future releases.

Human Interface Performance

Most command interpreters are concerned with two types of performance: response time and throughput. Response time is the amount of time between the entered command and completed response, and throughput is the number of commands per unit of time that can be completed. Response time is important when a user is interacting directly with the command interpreter, while throughput is most important when commands are processed from an OBEY file.

Startup time (the time from entering the program name to the time a prompt is issued), and thus response time, were optimized by accessing SCP only when it is needed. Throughput was optimized for OBEY file execution by using buffered edit file input and output, by providing a "no echo" option that suppresses the echo of the input file to the terminal, and by suppressing updates to the history buffer.

Conclusion

The implementations of SCP and SCF show that SPI is a good basis for network management products supporting large numbers of subsystems. Important programmatic interface goals, such as consistency, simplicity, and reliability, as well as human interface goals, such as maintainability, compatibility, and user friendliness, were all satisfactorily accomplished.

The case study of SCP and SCF not only highlights issues but also describes solutions, such as imposing additional programmatic and human interface standards, introducing a single subsystem control point (SCP), and a product module approach to the human interface (SCF). These issues and solutions apply to all implementations of programmatic and human interfaces where a large number of subsystems must be supported.

Tom Lawson received his education at California State University, Chico. He joined Tandem in 1984 as a network management software designer, has continued to work in the area of network management applications, and is currently the lead designer of SCF.

Using Subsystem Programmatic Interface and Event Management Services

Subsystem Programmatic Interface (SPI) is more conducive to automatically handling a diversity of systems and languages because the interface is programmatic rather than textual. Programmatic interfaces, however, are unfamiliar to many users and may appear more difficult to use than textual interfaces. This article tells the reader how to learn more about SPI and Event Management Service (EMS) and discusses some common issues, such as intrepeting numbers as named values and following semantic conventions, that arise when users debug an SPI program.

Before reading this article, the reader should be familiar with the information in *Introduction to Distributed Systems Management (DSM)*.

Methods of Learning SPI and EMS

The best way to become familiar with SPI and EMS is to attend a Tandem Education class and read the examples in the associated manuals. The manuals include the following types of examples:

- *Distributed Name Service (DNS™) Manual* gives essentially parallel examples for using SPI programmatically in COBOL, TAL, and TACL languages. Studying the DNS example gives insight into the ways in which the various languages deal with SPI.
- *DNS Manual* and *PATHWAY™ Management Programming Manual, Volume 2* give sample COBOL implementations.
- *Event Management Service (EMS) Manual* gives TAL and TACL examples for using SPI to communicate with a consumer distributor.
- *X25AM Management Programming Manual*, *FOX™ Management Programming Manual*, and *EXPAND™ Management Programming Manual* have complete TAL programs.

Because all of Tandem's implementations of SPI obey a standard, studying any of them will give general knowledge that can be used in dealing with any specific subsystem. The *EMS Manual* and the *EXPAND Management Programming Manual* interrelate, giving the reader an example of a management application for an EXPAND line.

Beyond reading the manuals and studying examples, users can write programs and experiment. Regardless of the language chosen, it is useful to set up a common framework (which allows easy modifications and additions) to study a prototype implementation of a DSM program. TACL can be used to interactively experiment with SPI, while the other languages require iterative edit, compile, and execute cycles.

Debugging SPI Programs

SPI introduces three major new features. SPI at its most basic level is a set of procedures for manipulating buffers, where the user establishes a set of token codes to describe the data in buffers. A second feature is the use of named values for variables that can take on only a small, fixed set of values; there are

named values for token numbers, commands, object types, event numbers, and other well-defined SPI entities. Finally, SPI requires a set of semantic conventions on the look and structure of requests, replies, and events by which all subsystems must abide. Tandem has provided the SPI procedures, named values for its subsystems, and a description of the conventions in its manuals.

The ability to interpret numbers as named values in a debugging environment presents new challenges. Mistakes made in following the semantic conventions also need to be found and then understood in the debugging environment. The remainder of this article gives examples of these two new aspects of debugging SPI programs.

Numbering Conventions

While creating an SPI program, the programmer uses many named values. In the debugging environment, these values, in their raw form, are numbers. During the debugging process, a programmer will frequently need to translate the numbers back into names to establish their meaning.

Several conventions must be understood to easily and correctly map a number back to its name. First, the owner of the value must be established. Usually, the owner of the value is the Subsystem Identifier (SSID) owning the token. Several shared values, however, may also appear as being owned by a particular subsystem when they are defined in common.

Commonly defined tokens may take their value either from the defining subsystem or from the subsystem that owns the buffer or token. For example, ZSPI defines the values for ZSPI-TKN-HDRTYPE, while the values of ZSPI-TKN-COMMAND are defined individually by the subsystems.

The general rule for token codes is that those with negative token numbers are commonly defined and those with positive token numbers are subsystem-specific. Currently, only ZSPI and ZEMS provide common negative token numbers. The Subsystem Control Process (SCP) common interface further reserves token numbers 4096 through 9999 for the ZCOM prefix.

The SCP common interface also defines several more conventions than provided by the more basic SPI conventions.

Figure 1

Summary for SCP common interfaces

Name in Value type	XXXXddl	zCOMddl	zSPIddl	zEMSddl
Token number	1 - 4999	4096 - 9999	0 - -512	-513 - -530
Command number		0 - 8191		
Object type number		0 - 4095		
Event number	0 - 9999	-1 - -9999		
Error number	0 - 9999	-1 - -9999		

Summary for non-SCP common interfaces

Name in Value type	XXXXddl	zSPIddl	zEMSddl
Token number	1 - 9999	0 - -512	-513 - -530
Command number	-9999 - 9999		
Object type number	-9999 - 9999		
Event number	-9999 - 9999		
Error number	-9999 - 9999		

Figure 1.
Summary for SCP and non-SCP common interfaces.

SCP/Non-SCP Comparisons

ZCOM enumerates values for commands, object types, some tokens, some events, some common errors, and other token values with identifiable common characteristics across subsystems. For subsystems not requiring the SCP, the enumerated values are unique to the subsystem. Thus, ZGDS and ZLAN share the ZCOM command numbers, while ZEMS and ZPWY have individually defined their own ZEMS and ZPWY command numbers.

Subsystems using SCP must abide by constraints on commands, object types, token numbers, error numbers, and event numbers. All commands (ZSPI-TKN-COMMAND, ZCOM-CMD-*xxx*) and object types (ZSPI-TKN-OBJECT-TYPE, ZCOM-OBJ-*xxx*) are defined by ZCOM rather than the qualifying subsystem. The same holds true for negative error numbers (ZSPI-TKN-RETCODE, ZCOM-ERR-*xxx*) and negative event numbers (ZEMS-TKN-EVENTNUMBER, ZCOM-EVT-*xxx*).

The summary for SCP common interfaces and for non-SCP common interfaces is shown in Figure 1.

Figure 2

```
?SYMBOLS, INSPECT, NOLIST, NOMAP, NOCODE

?NOLIST, SOURCE zspital
?LIST
?NOLIST, SOURCE zcomtal
?LIST
?NOLIST, SOURCE zscptal
?LIST

STRUCT .spibuf( ZSCP^DDL^MSG^BUFFER^DEF );
STRUCT .zscp^val^ssid( ZSCP^VAL^SSID^DEF );

?NOLIST, SOURCE extdecs( INITIALIZER, DEBUG, STOP
?                      , SSINIT, SSPUTTKN, SSNULL, SSPUT, SSGETTKN
?                      , OPEN, WRITEREAD, CLOSE )
?LIST
?PAGE

PROC spi^demo MAIN; -- Perform STATUS on SCP.
BEGIN
    INT scp;                                -- File number of scp.
    INT len;                                -- Number of bytes written and read.
    INT status;                             -- Status of SS call.
    INT retcode;                            -- Return code from server.

    zscp^val^ssid ':=' [ ZSPI^VAL^TANDEM, ZSPI^SSN^ZSCP, ZSCP^VAL^VERSION ];
    CALL INITIALIZER;

    spibuf ':=' "$ZNET    #ZSPI ";          -- Default name.
    CALL OPEN( spibuf, scp );
    IF <>THEN CALL DEBUG;

    IF status := SSINIT( spibuf, ZCOM^VAL^BUFLEN, ZSCP^VAL^SSID , ZSPI^VAL^CMDHDR,
                         ZCOM^CMD^STATUS , ZCOM^OBJ^PROCESS )
        THEN CALL DEBUG;

    IF status := SSGETTKN( spibuf, ZSPI^TKN^USEDLEN, len )
        THEN CALL DEBUG;

    CALL WRITEREAD( scp, spibuf, len, ZCOM^VAL^BUFLEN, len );
    IF <>THEN CALL DEBUG;

    len := ZCOM^VAL^BUFLEN; -- TAL requires variable for parameter
    IF status := SSPUTTKN( spibuf, ZSPI^TKN^RESET^BUFFER, len )
        THEN CALL DEBUG;

    IF status := SSGETTKN( spibuf, ZSPI^TKN^RETCODE, retcode )
        THEN CALL DEBUG;
    IF retcode <>ZSPI^ERR^OK THEN CALL DEBUG;

    CALL CLOSE( scp );
    CALL STOP;
END; -- spi^demo
```

Figure 2.
The TAL program getting STATUS of Subsystem Control Process (SCP).

Throughout the *Distributed Systems Management (DSM) Programming Manual* and the *Communications Manager Programming Manual* there are many semantic conventions. An example is the usage of ZSPI-TKN-RETCODE. The manual states that this token must be present in each response, and also that the value 0 means no errors have occurred. Mistakes that are commonly made when the semantic conventions are not followed are discussed later in this article.

Using the C10 Release Version of INSPECT

INSPECT, version C10, is the principal method for debugging SPI interfaces. An SPI buffer has an internal format that the user normally does not have to understand. If, however, the programmer uses DEBUG or earlier versions of INSPECT (before C10), there is no support for decoding an SPI buffer, and the user will have to manually decode the buffer (which assumes some knowledge of SPI internals).

The *INSPECT Manual* (with C10 update) presents the syntax of the new commands related to SPI. This article describes how to use these commands while debugging typical problems. Figure 2 shows a simple SPI program to get status information from the Subsystem Control Process (SCP).

INSPECT, version C10, supports a general method for displaying SPI buffers or individual tokens. Figure 3 shows a debugging session with the program resulting from the source in Figure 2.

In the debugging session, a display of the source code surrounding the CALL to DEBUG that caused INSPECT to appear is shown. There is an asterisk on line 60, which is the code immediately following the CALL DEBUG. A basic display of the RETCODE, which is tested in the IF statement, is made followed by a dump of the SPI buffer. INSPECT first displays the command/response buffer header tokens (all of the ZSPI^TKN^ items) followed by the tokens in the buffer. The buffer tokens are displayed with their SSID (e.g., TANDEM.SCP.C10), followed by their token code as a triplet of numbers (token data type, length, and number), followed by the token value.

Notice in the list of tokens codes in Figure 3 that the current SPI position (indicated by *) points to token code 11,2,0. Token number 0, according to the chart shown in Figure 1, should be looked up in zSPIddl where the name ZSPI-TNM-RETCODE is found. SSGET left the buffer positioned to the RETCODE token (type = 11, length = 2, number = 0).

ZSPI-TKN-RETCODE contains an error number, which in this case is negative. SCP negative error numbers are defined in zCOMddl. If looking up –29 as ZCOM-ERR-TKN-REQ in the zCOMddl file is not sufficient to indicate or describe the error, refer to the *Communications Management Programming Manual* for further details. This manual contains the definition of the error ZCOM-ERR-TKN-REQ ("a command request was issued in which a required token is missing") and a description of other tokens present in the error list.

To determine which token is missing, the token ZSPI-TKN-PARM-ERR must be located in the error list. This can be done by finding the token number (TNM) value of ZSPI-TNM-PARM-ERR as –250 in the zSPIddl file. The value of ZSPI-TKN-PARM-ERR is a struct called ZSPI-DDL-PARM-ERR that has a token code as its first element. The INSPECT output line of relevance is:

```
TANDEM.SCP.0 (7,8,–250)
   = ?25 " " ?255 ?13 ?0 ?1 ?0 ?0
```

INSPECT shows the values for 7,8,–250 as a list of bytes since its token type (7) is a struct of unknown attributes to INSPECT. The first byte of the token code field is the token data type and the second byte is the token data length. The third and fourth bytes (?255 ?13) give the token number of the missing token. The token number to look for is –243 (255 × 256 + 13 – 65536).

Alternatively, as shown at the end of Figure 3, INSPECT can be used to present the value of PARM-ERR in a more meaningful manner using the DISPLAY command to show an individual token as reproduced below:

```
-SPIDEMO-DISPLAY spibuf:zspi^tkn^parm^err
   POSITION zspi^tkn^errlist
   AS zspi^ddl^parm^err^def
```

Figure 3

```
13>RUN spidemo
INSPECT - SYMBOLIC DEBUGGER - T9673C00 - (15JUL87) SYSTEM \COMM
COPYRIGHT TANDEM COMPUTERS INCORPORATED 1984, 1985, 1986, 1987
INSPECT P=000166, E=000227
*116,01,088* SPIDEMO #SPI^DEMO + %163I SPIDEMOS
-SPIDEMO-SOURCE
     56  IF status := SSGETTKN( spibuf, ZSPI^TKN^RETCODE, retcode)
     57      THEN CALL DEBUG;
     58  IF retcode <>ZSPI^ERR^OK THEN CALL DEBUG;
     59
*    60  CALL CLOSE( scp );
     61  CALL STOP;
     62  END; -- spi^demo
-SPIDEMO-DISPLAY retcode
   RETCODE = -29
-SPIDEMO-D spibuf TYPE SPI-NUM
ZSPI^TKN^HDRTYPE = 0 (ZSPI^VAL^CMDHDR)
ZSPI^TKN^CHECKSUM = 0
ZSPI^TKN^COMMAND = 8
ZSPI^TKN^LASTERR = -8 (ZSPI^ERR^MISTKN ) - Token not found
ZSPI^TKN^LASTERRCODE = 620822276 (37,0,-252)
ZSPI^TKN^MAX^FIELD^VERSION = 0
ZSPI^TKN^MAXRESP = 0
ZSPI^TKN^OBJECT^TYPE = 17
ZSPI^TKN^SERVER^VERSION = 17162 C10
ZSPI^TKN^SSID = TANDEM.25.C10
ZSPI^TKN^USEDLEN = 138
BUFFER LENGTH = 900

TANDEM.SCP.C10          (11,2,9988) = 17
TANDEM.SCP.C10          (1,255,9990) - LENGTH 5 = "$ZNET"
TANDEM.SCP.C10          *(11,2,0) = -29
TANDEM.SCP.C10          (37,0,-252) - ERROR LIST
TANDEM.SCP.0              (11,2,9988) = 17
TANDEM.SCP.0              (1,255,9990) - LENGTH 5 = "$ZNET"
TANDEM.SCP.0              (28,14,-251) = TANDEM.SCP.C10 -29
TANDEM.SCP.0              (7,8,-250) = ?25 " " ?255 ?13 ?0 ?1 ?0 ?0
TANDEM.SCP.0              (39,0,-254) - END LIST
-SPIDEMO-D spibuf:zspi^tkn^parm^err POSITION zspi^tkn^errlist
        AS zspi^ddl^parm^err^def
  ZSPI^DDL^PARM^ERR^DEF =
    Z^TOKENCODE =
      Z^TKN =
        Z^DATATYPE = ?25
        Z^BYTELEN = " "
        Z^NUMBER = -243
    Z^INDEX = 1
    Z^OFFSET = 0
-SPIDEMO-
```

Figure 3.
INSPECT, version C10, displaying an SPI buffer from SPIDEMO program.

The use of the POSITION option allows access to tokens inside the error list to find the PARM^ERR token. The AS option tells INSPECT how to display the structure instead of its default of a list of bytes as shown previously. The relevant item displayed is the token number:

```
Z^NUMBER =  -243
```

Since the token number is in the range -1 to -512, -243 can be found in the file ZSPIDDL with the name ZSPI-TNM-MANAGER. The error list thus reveals that the source of the error is that the ZSPI-TKN-MANAGER token was not provided. A semantic convention is that all subsystems using ZCOM values require either ZSPI-TKN-MANAGER or ZCOM-TKN-OBJNAME to be present in all requests. The program in Figure 2 violated this and, as a result, got an error.

If the code

```
STRING .objname
   [0:ZCOM^VAL^OBJNAME^LEN + 1 ];
   -- object name
```

is added in the variable declarations (OBJNAME^LEN plus 2 bytes to hold the length), and if the code

```
objname ': = ' [ 0, 5, "$ZNET" ];
   -- 2 byte length and name

IF status
   : = SSPUTTKN( spibuf
      , ZCOM^TKN^OBJNAME , objname )
   THEN CALL DEBUG;
```

were added after the call to SSINIT, the program will then run to completion without error.

Typical SPI and EMS Mistakes

To use SPI and EMS, the many guidelines documented in the manuals must be properly implemented. Failure to follow a guideline may result in difficulties determining the cause of problems.

Initializing Subsystem SSIDs. An SSID must be explicitly initialized in both TAL and TACL. DDL (Data Definition Language) does not generate files for those languages that provide the initialization as specified by the VALUE clause in DDL. If the SSID definitions that are provided in the *xxxx*COB files are not used, COBOL programs will only see uninitialized SSIDs. If the SSID is not initialized, error -5, missing parameter, is returned from SSINIT. One may wonder what is missing from the SSINIT call since all the correct parameters were supplied. The SSID parameter is considered missing if its value is 0. Typically, if the SSID is not initialized, its value will be 0 and will be considered missing by SSINIT.

SSGET Index 1. Another common mistake is to forget to specify the INDEX parameter to SSGET. It is common practice to supply an INDEX of 1 with each call to SSGET. This is because the order in which tokens will occur in the reply buffer is never guaranteed, yet SSGET with INDEX 0 (the default) implies that the token expected is after the current one. So, if error -8, missing token, is returned from SSGET, make sure INDEX 1 is being used.

Qualify Tokens with the Correct Subsystem. Associated with all token codes is an implicit (the default) or explicit subsystem identifier. The semantic convention for command buffers is that all the tokens in the request should be qualified by the subsystem specified in the header. Similarly, all tokens in a response buffer, with the exception of error lists, should be qualified by the subsystem specified in the header. However, event buffers may freely use tokens from other subsystems.

With event buffers, it is easy to forget that the name of a token, such as ZEMS-TKN-TEXT, means nothing to EMSADDTOKENS (or EMSGET) without specifying the subsystem that the token is qualified by. The procedures see only a number like 33685518. From

the number, the procedures have no way of knowing if the number represents ZPWY-TKN-DEVICESUBTYPE or ZEXP-TKN-LH-ERR-NUM since they both have that value. The procedures must be explicitly told what subsystem qualifies a token.

The current guidelines on token qualification for C00 are:

- ZSPI. All ZSPI tokens should be PUT/GET without qualification, with the occasional exception of ZSPI-TKN-ERRLIST. When ZSPI-TKN-ERRLIST is qualified by an SSID other than the default, it is called a foreign error list.
- ZCOM. All ZCOM tokens are implicitly qualified by the subsystem using them.
- ZEMS. ZEMS tokens, with negative token numbers (common tokens) should not be qualified, but ZEMS must be the qualifier of all ZEMS tokens with non-negative token numbers (subsystem-specific).

Most negative token numbers are header tokens for which qualification is ignored. The following are buffer tokens qualified by the default SSID:

ZEMS-TKN-LDEVNAME
ZEMS-TKN-SUBJECT

All positive tokens must be qualified by ZEMS regardless of the default SSID to be regarded as EMS tokens. The following tokens are typically used in event buffers and require ZEMS qualification:

ZEMS-TKN-TEXT
ZEMS-TKN-ACTION-ID
ZEMS-TKN-ACTION-NEEDED
ZEMS-TKN-CU
ZEMS-TKN-LDEV

Qualify ZEMS-TKN-TEXT by ZEMS. If an EMS buffer is initialized with a non-ZEMS SSID (such as XOUR-VAL-SSID) and ZEMS-TKN-TEXT is added via EMSADDTOKENS, the event will never be printed with the text. Only ZEMS-TKN-TEXT tokens qualified by ZEMS are printed. If the text of the event is to be printed, ZEMS-VAL-SSID (previously initialized) must be specified along with ZEMS-TKN-TEXT on a EMSADDTOKENS call.

Testing Software Using SPI and EMS

Since SPI and EMS rely on semantic conventions, it is imperative that programs using them be tested for conformance to the conventions. Without specific attention, the common errors discussed above can easily slip through the debugging phase.

The EMSADDTOKENS procedure treats a zero-value SSID as an unspecified parameter. Since an uninitialized SSID is typically 0, EMSADDTOKENS will ignore the SSID and use the default SSID. Thus, when testing that a subsystem builds its events correctly, the owner of each token should be verified.

The semantic convention of order independence of tokens should be verified. When testing a server, the same request with tokens in different orders should be sent to the server. When testing a requester, a dummy server that mixes the order of returned tokens should be used. If each token is the first token during some request or reply, it is unlikely the server has dependencies on token ordering.

Figure 4

```
FILTER check-emphasis;
-- Returns PASS 0 if XOUR event has expected emphasis.
--         PASS 1 if XOUR event was emphasized when
--                it should not have been emphasized.
--         PASS 2 if XOUR event was not emphasized when
--                it should have been.
BEGIN
   IF ZSPI^TKN^SSID <>SSID( xour^val^ssid ) THEN FAIL;
   IF ZEMS^TKN^EMPHASIS = 0 THEN
      IF ZEMS^TKN^EVENTNUMBER = [xour^evt^notemph^1]
      OR ZEMS^TKN^EVENTNUMBER = [xour^evt^notemph^2]
         -- OR ...
      OR ZEMS^TKN^EVENTNUMBER = [xour^evt^notemph^N]
      THEN PASS 0 ELSE PASS 1;
ELSE
      IF ZEMS^TKN^EVENTNUMBER = [xour^evt^emph^1]
      OR ZEMS^TKN^EVENTNUMBER = [xour^evt^emph^2]
         -- OR ...
      OR ZEMS^TKN^EVENTNUMBER = [xour^evt^emph^N]
      THEN PASS 0 ELSE PASS 2;
END;
```

Figure 5

```
FILTER checkemsevents;
-- Returns PASS 0 if event has expected tokens.
--         PASS 1 if event is missing tokens.
BEGIN
   IF ZSPI^TKN^SSID <>SSID( zems^val^ssid ) THEN FAIL;
   IF ZEMS^TKN^EVENTNUMBER = [ZEMS^EVT^FILESWITCH]THEN
      BEGIN
      IF ZEMS^TKN^SUBJECT = [ZEMS^TKN^COLLECTOR]
      AND TOKENPRESENT( ZEMS^TKN^COLLECTOR )
      AND TOKENPRESENT( ZEMS^TKN^LOGSWITCHREASON)
      AND TOKENPRESENT( ZEMS^TKN^LASTLOGFILE )
      AND TOKENPRESENT( ZEMS^TKN^NEWLOGFILE )
      AND TOKENPRESENT( ZEMS^MAP^COL^STATUS )
      THEN PASS 0 ELSE PASS 1;
      END;
   IF ZEMS^TKN^EVENTNUMBER = [ZEMS^EVT^FILE^ROTATE^PURGE]THEN
      BEGIN
      IF ZEMS^TKN^SUBJECT = [ZEMS^TKN^COLLECTOR]
      AND TOKENPRESENT( ZEMS^TKN^COLLECTOR )
      AND TOKENPRESENT( ZEMS^TKN^PURGEDLOGFILE)
      THEN PASS 0 ELSE PASS 1;
      END;
   -- IF ...
END;
```

Figure 4.
EMS filter checking for EMPHASIS of user subsystem events.

Figure 5.
EMF filter verifying presence of tokens in EMS subsystem events.

When a new subsystem server is being tested, the first tests can be made without knowing anything about the specifics of its SPI implementation. The semantic conventions require that all subsystems support command number 0 as the GETVERSION command. Furthermore, servers are required to properly respond to various values of MAXRESPONSE. Thus, three different GETVERSION requests with MAXRESPONSE 0, 1, and -1 can be sent to the server. The first should return a RETCODE not in a data list, while the other two should enclose the response in a data list. Similarly, it can be tested if a subsystem detects illegal tokens, subsystem ID, or object type using a GETVERSION command.

Using EMS Distributors and Filters to Test EMS Events

With an EMS filter, a fast check for the presence of all tokens expected in an event can be made. If the contents of those tokens are constant, the token values are easy to verify. It is possible to specify the expected values for tokens with varying values.

Starting with a simple case, suppose one wanted to make sure that the EMPHASIS token was correctly set for each event generated. The filter in Figure 4 could be created.

An application similar to the one given in Appendix C of the EMS manual can now use a consumer distributor to filter events. If the PASS value was not 0, it would print the event number of the offending event and what emphasis was expected.

The VIEWPOINT™ Alternate Events screen can also be configured to use the filter. Events with missing emphasis will show up as ACTION events (pass value 1) and events without emphasis will show up as CRITICAL events (pass value 2). VIEWPOINT will add up the number of unacceptable events.

More powerful filters can be written to verify that events contain all expected tokens. For example, if it was desired to check all the documented tokens were present for EMS events, the filter shown in Figure 5 could be used. The filter can be compiled with the statements shown below:

#PUSH *list*
#LOAD/LOADED *list/* ZSPIDEF.ZSPI
#LOAD/LOADED *list/* ZSPIDEF.ZEMS
EMF/IN emsfs/ emsfo

Note that ZEMS^TKN^SUBJECT is a derived token whose value is the token following the ZEMS^TKN^SUBJECTMARK. Thus, using ZEMS^TKN^SUBJECT not only ensures that there is a SUBJECTMARK but also that it is in the correct place (i.e., preceding the subject of the event).

SUBJECTMARK is the only positional token defined in Tandem SPI and EMS interfaces. Note that the order of locating tokens is immaterial. For instance, LOGSWITCH-REASON is retrieved near the beginning of the filter source even though in C00 it actually occurs at the end of the buffer.

VIEWPOINT can be used again as the application that deals with this filter. ACTION events would be those with missing tokens.

A more complex filter to test an event would verify not only that the tokens are present, but also that they have the correct values. To check for the actual token values, a parameterized filter is needed so that the expected values for tokens that don't have fixed values can be dynamically specified. Figure 6 shows the start of such a parameterized filter for the EMS events given in Figure 5.

It will probably be easier to write custom application code to test for token values than to create a customized filter as shown in Figure 6. With the parameterized filter, the application must create DDL for the parameter tokens and then add the parameter tokens and their values to a CONTROL command to the distributor. The verification of token values could be broken up between filter and application. A filter more powerful than that shown in Figure 5, but without the parameters of the filter in Figure 6, could test that all expected tokens are present and those with fixed values, such as CONSOLE-PRINT and EMPHASIS, have the correct values.

Figure 6

```
FILTER check_ems_events(
        SSID( test^val^ssid, P514^TKN^LOGSWITCHREASON ),
        SSID( test^val^ssid, P514^TKN^LASTLOGFILE ),
        SSID( test^val^ssid, P514^TKN^NEWLOGFILE ),
        SSID( test^val^ssid, P514^MAP^COL^STATUS ),
        SSID( test^val^ssid, P520^TKN^PURGEDLOGFILE ));

-- Returns PASS 0 if event has expected tokens.
--         PASS 1 if event is missing tokens.
BEGIN
    IF ZSPI^TKN^SSID <>SSID( zems^val^ssid ) THEN FAIL;
    IF ZEMS^TKN^EVENTNUMBER = [ZEMS^EVT^FILESWITCH]THEN
        BEGIN
        IF ZEMS^TKN^SUBJECT                 =[ZEMS^TKN^COLLECTOR]
        AND ZEMS^TKN^CONSOLE^PRINT          <>0 -- TRUE
        AND ZEMS^TKN^EMPHASIS               <>0 -- TRUE
        AND ZEMS^TKN^COLLECTOR              = [ZEMS^SUBJ^PCOLL]
        AND
        ZEMS^TKN^LOGSWITCHREASON            = P514^TKN^LOGSWITCHREASON
        AND ZEMS^TKN^LASTLOGFILE            = P514^TKN^LASTLOGFILE
        AND ZEMS^TKN^NEWLOGFILE             = P514^TKN^NEWLOGFILE
        AND ZEMS^MAP^COL^STATUS             = P514^MAP^COL^STATUS
        THEN PASS 0 ELSE PASS 1;
        END;
    IF zems^tkn^eventnumber = [ZEMS^EVT^FILE^ROTATE^PURGE]THEN
        BEGIN
        IF ZEMS^TKN^SUBJECT                 = [ZEMS^TKN^COLLECTOR]
        AND ZEMS^TKN^CONSOLE^PRINT          <>0 -- TRUE
        AND ZEMS^TKN^EMPHASIS               <>0 -- TRUE
        AND ZEMS^TKN^COLLECTOR              = [ZEMS^SUBJ^PCOLL]
        AND ZEMS^TKN^PURGEDLOGFILE          = P520^TKN^PURGEDLOGFILE
        THEN PASS 0 ELSE PASS 1;
        END;
    -- IF ...
END;
```

Figure 6.
EMS filter verifying values of tokens in EMS subsystem events.

Conclusion

This article has shown that with the advent of the new SPI and EMS technology, new debugging and testing techniques have become necessary. INSPECT has been enhanced in release C10 to provide some of the new techniques. Knowing how to translate numbers back into names in the debugging environment is necessary to understand problems that arise. Problems most frequently are caused by failure to follow semantic conventions. Because of the semantic conventions, testing takes on increased importance. EMS filters are a tool that can help test EMS events.

References

Communications Management Programming Manual. Part no. 84110. Tandem Computers Incorporated.

INSPECT Manual. Part no. 84149. With *Update 1*. Part no. 11160. Tandem Computers Incorporated.

Distributed Name Service (DNS) Manual. Part no. 84093. Tandem Computers Incorporated.

Distributed Systems Management (DSM) Programming Manual. Part no. 82587. Tandem Computers Incorporated.

PATHWAY Management Programming Manual, Volume 2. Part no. 84113. Tandem Computers Incorporated.

Event Management System (EMS) Manual. Part no. 84092. Tandem Computers Incorporated.

EXPAND Management Programming Manual. Part no. 84109. Tandem Computers Incorporated.

FOX Management Programming Manual. Part no. 84136. Tandem Computers Incorporated.

Introduction to Distributed Systems Management (DSM). Part no. 84091. Tandem Computers Incorporated.

X25AM Management Programming Manual. Part no. 84135. Tandem Computers Incorporated.

Keith Stobie joined Tandem in 1979 and was the lead QA person for the DSM project with specific responsibility for the EMS Collector. Previously he worked on testing TMF and TRANSFER and was an original member of the Gremlin group.

Estimating Host Response Time in a Tandem System

The definition of response time depends upon who is using the term. Most often, user response time is meant. In essence, user response time is the time between keyboard lock and keyboard unlock (i.e., the time spent sending the request to the computer, the time the computer spent processing the request, and the time spent sending the response back to the user's terminal). User response time is also referred to as network response time. The time the computer system spends processing the request is referred to as the host response time. This article addresses estimating the host response time within a Tandem system.

The article begins with a discussion of the mathematical theory required to understand the analytical modeling of host response time. It then describes the impact different disk configurations have on response time and how this is incorporated into an analytical model. The article concludes with an example of how an analytical model of response time is built, with a comparison between the estimated response time from the model and the actual measured response time.

Analytical Modeling of Response Time

Ideally, if a transaction is the only work in the system, the host response time would be equal to the time required to service that transaction. The transaction would not be competing with other work for resources and, therefore, would not incur any delays waiting for a particular resource to free up. In an on-line system, multiple transactions are being entered at any given time. These transactions are competing for resources such as the CPU, the device controllers, and the actual devices. Sometimes a transaction is forced to wait for a resource to become free and a queue forms for that resource.

Queueing theory is the field of mathematics that studies the behavior of waiting lines. Much work has been done modeling computer systems as sets of waiting lines. Buzen and Denning (1978) used the foundations of queueing theory to derive the techniques of modeling computer systems known as Operational Analysis. Operational Analysis is based on the fact that, in a given measured interval, certain operational laws hold true on the relationships between measured quantities irrespective of the statistical characteristics of the data (*ACM Computing Surveys*, 1978).

Five quantities define a queueing system:

1. Distribution of time between customer arrivals. (In a computer system, a transaction is viewed as the customer.)
2. Statistical distribution of service time.
3. Number of servers available.
4. Maximum capacity within the queue.
5. Selection criteria for the next customer to be serviced.

David Kendall's method of notation has become the standard for describing a queueing system. A queueing system is represented as 1/2/3/4/5 where the numbers are associated with the above list. For the two statistical distributions (numbers 1 and 2), standard conventions are used (Allen, 1978):

M – exponential distribution
E – Erlang-K distribution
D – deterministic (constant)
.
.(etc.)

If the last two positions are not present, they take on the defaults of an infinite size queue and a first come/first served (FCFS) queue discipline, respectively.

For example, a queue described as M/M/3 means that the interarrival time of transactions is exponentially distributed; the service time is exponentially distributed; there are three servers; the queue is infinite in potential size; and the arrivals are serviced on a FCFS basis.

Transactions in an on-line transaction processing (OLTP) system arrive randomly. Because of its pleasant mathematical properties, the exponential pattern is most commonly assumed for queueing theory models. The service time pattern of these arrivals are often described by the exponential distribution as well. This is because of "the Markov or memoryless property of the exponential distribution which allows the model to imply that the amount of time remaining to complete a customer's (i.e., a transaction's) service is independent of the service time already provided" (Allen, 1978).

A Tandem host system can be viewed as two "service centers," each having its own service time pattern. (A service center is a place where a transaction requires some amount of service or demand.) The balanced CPUs[1] within a Tandem node can be viewed as a single service center with a single server of capacity $N \times 100\%$, where N is the number of CPUs in the system and 100% represents the maximum amount of one CPU that is available. If balanced, the disks can also be viewed as a single service center with a single server of capacity $M \times 100\%$, where M is the number of disk drives. By considering the two service centers as single server service centers, the formulas based on an M/M/1 queueing model can be used. The system must be balanced and well tuned in order to view each group as a single server.

A properly tuned system does not contain any internal queues (such as queues at server processes) that affect the transaction's time in the system. The only delay incurred is the time waiting to enter the service center. Once a transaction enters the service center, there are no other queues in its path.

The following formula is used to calculate the response time for an M/M/1 service center:

$$R = \frac{D}{1 - U}$$

where

D = time demand at the service center

U = utilization (percent busy) of the service center

The busier the service center (i.e., as U approaches 1), the longer the response time.

For the purpose of defining a response-time model, a Tandem™ system can be viewed as two distinct service centers: CPUs and disk drives. Each needs to be analyzed separately before the response times are added together. Separate analysis is necessary because typically a transaction is either in the CPU or at the disk, but not both at the same time. (This is not true for a system where no-waited I/O is taking place.)

[1]Balanced CPUs means that the utilization of each CPU is close to the utilization of the other CPUs. The same holds true for disks; i.e., balanced disks means that the utilization of each disk is close to the utilization of the other disks.

The response-time estimate for the two service centers is :

$$R(total) = R(cpu) + R(disk)$$

$$= \frac{D(cpu)}{1 - U(avg\ cpu)} + \frac{D(disk)}{1 - U(avg\ disk)}$$

The technique used to determine the demand of a given transaction at the CPU or disk subsystem is called consumption modeling (Horwitz, Shugh, and Sitler, 1988).

After the system is balanced and well tuned, it is measured for the purposes of constructing a baseline model. A baseline measurement consists of both a measurement of the system and a measurement of the response time. By having the actual response time, a model can be created that provides a response-time estimate; that estimate can then can be compared to the actual response time measured. This is the first step in validating the model.

When the two response times are compared, the estimated response time may not be exactly equal to the actual response time measured. In this case, the model needs to be calibrated to match reality. One method is to incorporate the error in the baseline model into the forecasts. This technique is known as a relative change model. The response-time formula for the relative change model is:

$$A = B \times C$$

where

A = modified response-time estimate

B = response time estimated by model

$C = \frac{\text{actual}}{\text{estimated}}$ baseline response time

In essence, the formula assumes that the percentage the estimated response time is off for the baseline transaction rate will be the percentage the estimates will be off using the model for other transaction loads.

The "correction factor" is the ratio of the actual response time to the estimated response time in the baseline model. It is the number by which the baseline estimate must be multiplied to produce the measured response time as the result.[2] (Refer to C in the preceding formula.)

Serial vs. Parallel Writes

There is a difference in estimating response time in a system consisting of mirrored volumes configured for serial writes as opposed to volumes configured for parallel writes. The physical configuration of a mirrored disk system is the primary factor affecting the response-time estimates of the disk subsystem using consumption modeling.

To simplify the modeling process on the first cut, an assumption is made that each logical I/O in the system results in the same number of physical I/Os.[3] This may not be a valid assumption for a system. A system where one application uses only key-sequenced files and another application uses only unstructured files is one example where further disk analysis would be necessary. If a simple model can achieve the level of accuracy desired, a more sophisticated version may not be necessary.

With serial writes, when a transaction performs a write operation, it must wait for both halves of the mirror to be written to before it can continue processing. As the name implies, these writes are done one at a time. The response time for the transaction is dependent on the completion of both the primary write and the mirror write.

[2] There are bounds on the transaction rate for which this model will hold. The farther away (either larger or smaller) from the baseline transaction rates the estimate is being made, the less accurate the model will be. Any model that is used for future predictions needs to be revisited frequently.

[3] A logical I/O is a request (Read, Write, Update, Delete) sent from an application to the disk processes. A system I/O is the total number of different records that had to be fetched in order to satisfy a logical request. This equals cache hits plus physical I/Os. A physical I/O is the total number of actual reads and writes on the device.

In the parallel write configuration, the writes to the two halves of the mirror are done in parallel; the transaction is only waiting for one of the two writes to complete. A distinction then becomes necessary between the physical disk demand needed for determining disk utilization requirements, and the physical disk demand needed for response-time estimation. For determining disk utilization, it is necessary to consider both halves of the mirrored volume as demand. For estimating response time, it is necessary to subtract the part of the disk utilization that can be attributed to the second write from the total disk utilization. This must also include any seek time associated with the write as well.

MEASURE™ (a performance statistics gathering facility) provides good metrics for calculating the disk demand for response time. The first step is to separate the read activity from the write activity (in each case including its share of the seek activity) for each spindle configured for parallel writes. It is assumed that seeking is done for all requests, and thus, an average seek time is associated with each physical I/O.

The ratio of the rate of the particular activity to the total read and write rate is used to determine the seek activity associated with either the reads or writes. Multiply the SEEK BUSY on the spindle by this percentage. The result is the amount of the seek activity that is to be grouped along with either the READ BUSY or WRITE BUSY. Figure 1 shows how this is expressed mathematically.

The write activity (TOTAL WRITE BUSY) for the mirrored half with less activity is not used to calculate the disk demand for estimating response time. This approach is somewhat pessimistic and assumes that the transaction would always have to wait for the longer of the two writes (primary and mirror) to complete. The difference between the disk utilizations of the primary and the mirror are usually not substantial enough to skew the results.

The physical disk demand per transaction for the response-time calculations can now be found by using a modified total disk utilization in the consumption model. The modified total disk utilization for response-time estimation is represented mathematically as:

Modified total disk utilization
(for parallel write configuration)
= Total disk utilization
– (sum of the lowest write activity
for each mirrored parallel pair)

Figure 1

$$\text{Read \% of seeking} = \frac{\text{Read rate}}{\text{Read rate} + \text{Write rate}} \times \text{Seek busy}$$

$$\text{Total read busy} = (\text{Read \% of seeking}) + \text{Read busy}$$

$$\text{Write \% of seeking} = \frac{\text{Write rate}}{\text{Read rate} + \text{Write rate}} \times \text{Seek busy}$$

$$\text{Total write busy} = (\text{Write \% of seeking}) + \text{Write busy}$$

Figure 1.
Apportioning seek activity to read and write activity.

Forecasting Response Times

Forecasting the new demand in a system given a change in the transaction rate is discussed in detail in the technical paper, "Performance Management and SURVEYOR" (Horwitz, Shugh, and Sitler, 1988). These results can be used with response-time estimating techniques to derive an estimate of the expected response time given the new demands on the system. This estimate can be used to determine whether service level objectives will be met and, if not, provide insight into what will be needed to achieve them.

The accuracy of the new estimate relies on the accuracy of the model and the linearity of the system being modeled. The model cannot predict any bottlenecks within the application (or system) that would add additional time to the transaction's response. The new estimate should be viewed as just that, an estimate. At best, it can be considered in terms of magnitude; i.e., will the new demand cause the response time to change from subsecond to multisecond if the system hardware remains the same.

The following example demonstrates forecasting techniques by taking a baseline model and predicting the response time at both higher and lower transaction rates. This includes modifying the estimate using the relative change prediction technique and comparing the results of the two estimates.

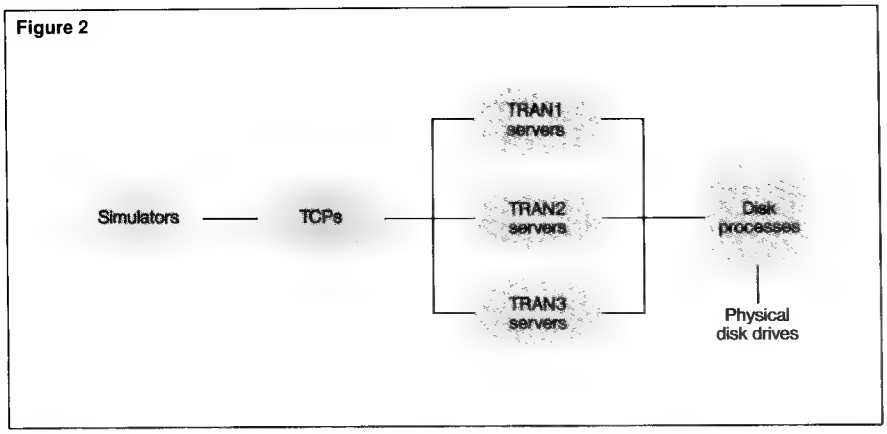


Figure 2.
Logical view of example PATHWAY system.

Creating a Response-Time Model—An Example

The following example shows how to estimate response time using an M/M/1 approach as well as how the relative change prediction affects the results. The system being analyzed consists of one PATHWAY™ (transaction processing system) application, which is run through the use of terminal simulators. Terminal simulators are processes that run within the same system as the PATHWAY application and are defined as terminals in the PATHWAY startup files. They are opened by the Terminal Control Process (TCP). These processes send messages to the PATHWAY application as if they were a user at a terminal. The message format is fixed for this application, but the contents of the message are random.

The simulator processes record statistics in a buffer that are periodically written out to a file. The response time is one of the statistics maintained. This permits a comparison of the estimate the response-time model makes to the actual response time achieved, to come up with the correction factor. The transaction rates within the system are configurable within the simulators (through the use of a parameter to the simulator process). There is one of these simulator processes running for each terminal in the PATHWAY system.

The PATHWAY application consists of three transaction types that differ in processing time and I/O intensity. For the sake of differentiation, they will be referred to as TRAN1, TRAN2, and TRAN3. Each transaction type is serviced by a separate server class. A logical view of this application is shown in Figure 2.

The hardware comprises four TXP™ processors with four mirrored disk drives. The database is spread across all four disks. MEASURE is used to gather measurement data from the CPUs, disks, and processes. The measurements were taken after the initial PATHWAY startup so that the system was in operating mode (i.e., steady state).

One series of tests was run with the disk drives configured for parallel writes. A baseline test was run at a transaction rate that ensured the system was not stressed (between 40-55% average CPU utilization). A consumption model was built from this baseline test and the response-time estimate for each transaction type was calculated from the M/M/1 formula. Five other transaction rates were then estimated using both the M/M/1 formula and the relative change technique. These transaction rates were then run in the system and the actual results were compared to those predicted with the models.

The analysis of the baseline model includes the following steps:

- Calculating the physical disk demand to be used in the response-time formula (demonstrated on one drive).
- Creating the relative change correction factor.
- Using the baseline model and the correction factor to forecast response times at a different transaction rate.

Baseline Model

CPU Service Center. Table 1 shows the transaction rates, the CPU demand per transaction (determined by consumption modeling), the logical I/Os per transaction, and the actual response time as reported by the simulators. The CPU demand per transaction includes TCP demand, server demand, and disk process demand. The simulator's demand per transaction was 8.0 ms. The simulator's demand per transaction is not included in the CPU demand for the transactions in Table 1 because the response time reported by the simulator process does not include the transaction's time within the simulator.

In consumption modeling, the interrupt handling in the baseline model is represented as a percentage of the total process busy calculated from all the identified workloads. This is necessary in order to be able to come up with an estimate of the interrupt handling that will be present with the new workload demands. The point of view taken is that the workload processing is responsible for the interrupt work. As the workload increases, so will the interrupt processing. For the example, the PROCESS BUSY due to workloads is 191.53%. The total CPU busy is 229%. Therefore, interrupt processing accounts for 37.47% (229 – 191.53). In terms of the PROCESS BUSY, 37.47 is 19.6% of 191.53.

Table 1.
Baseline model parameters.

Transaction type	Transaction rate	CPU D per transaction (ms)	Logical I/O per transaction	Actual response time (ms)
TRAN1	2.11	147.53	6.19	500
TRAN2	3.46	63.07	1.00	110
TRAN3	12.20	101.89	3.04	300
Simulators	17.77	8.00	N/A	N/A

Figure 3

$$TRAN1\ R(CPU) = \frac{D(CPU)}{1 - U(average\ CPU)} = \frac{147.53}{1 - 0.5725} = 345.10\ ms$$

$$TRAN2\ R(CPU) = \frac{D(CPU)}{1 - U(average\ CPU)} = \frac{63.07}{1 - 0.5725} = 147.53\ ms$$

$$TRAN3\ R(CPU) = \frac{D(CPU)}{1 - U(average\ CPU)} = \frac{101.89}{1 - 0.5725} = 238.34\ ms$$

The total CPU utilization for this system was 229% for four CPUs, giving an average CPU utilization of 57.25% (229/400). All the information is now available to obtain the CPU subsystem's contribution to response time as shown in Figure 3.

Figure 3.
Calculation of CPU subsystem response time.

Table 2.
Disk metrics.

Disk name	Read utilization	Write utilization	Seek utilization	Read rate	Write rate	Seek rate
$SYSTEM-P	4.27	6.75	5.10	2.89	4.69	6.33
$SYSTEM-M	8.90	6.52	6.59	6.00	4.69	7.90
$DATA-P	3.92	7.99	7.49	2.68	5.55	7.00
$DATA-M	8.87	7.76	9.20	5.97	5.55	8.85
$D1-P	3.65	6.51	4.27	2.49	4.50	5.81
$D1-M	8.06	6.24	6.10	5.46	4.50	7.23
$D2-P	4.84	7.08	6.22	3.29	4.90	7.14
$D2-M	8.60	6.96	7.40	5.84	4.90	8.53

Table 3.
Disk drive results.

Disk drive	Total disk busy (R + W + S)	Total write/seek busy PRIMARY	Total write/seek busy MIRROR	Lower WRITE busy
$SYSTEM	38.10	9.91	9.41	9.41
$DATA	45.23	13.24	12.19	12.19
$D1	34.83	9.26	9.00	9.00
$D2	41.10	10.80	10.34	10.34

Disk Drive Service Center. Because the disk drives are configured for parallel writes, the portion of disk demand that is due to the mirrored write needs to be determined and then removed from the disk demand for the transaction. This portion of demand is included in the calculation of average utilization because the transaction still has to compete with the mirrored write for access to a disk drive. Table 2 lists those metrics from MEASURE (system performance measurement tool) for the system under test that are used for analyzing the disk subsystem.

The calculations shown in Figure 4 are for the disk $D2. These same calculations were performed on all four drives. The results of these calculations for all the drives are given in Table 3.

To determine the portion of disk utilization to include in the demand per transaction for response-time estimation, proceed as follows:

1. Sum the lowest total write busy for each set of disks (Table 3, last column).
2. Subtract the result from the sum of the total disk busy (Table 3, second column) for all the disks.

The remainder is then the amount of disk utilization that is apportioned over all the logical I/Os for all the transactions.

From the measurement information, the number of logical I/Os per second is 55.36, of which 53.61 is a result of the application processing. MEASURE and the simulators account for the additional 1.75 logical I/Os per second.

Figure 4

For the primary drive:

$$\text{Read \% of seeking} = \frac{\text{Read rate}}{\text{Read rate + Write rate}} \times \text{Seek busy}$$
$$= \frac{3.29}{3.29 + 4.90} \times 6.22\%$$
$$= 2.50\%$$

$$\text{Total read busy} = (\text{Read \% of seeking}) + \text{Read busy}$$
$$= 2.50\% + 4.84\%$$
$$= 7.34\%$$

$$\text{Write \% of seeking} = \frac{\text{Write rate}}{\text{Read rate + Write rate}} \times \text{Seek busy}$$
$$= \frac{4.90}{3.29 + 4.90} \times 6.22\%$$
$$= 3.72\%$$

$$\text{Total write busy} = (\text{Write \% of seeking}) + \text{Write busy}$$
$$= 3.72\% + 7.08\%$$
$$= 10.80\%$$

For the mirror drive:

$$\text{Read \% of seeking} = \frac{\text{Read rate}}{\text{Read rate + Write rate}} \times \text{Seek busy}$$
$$= \frac{5.84}{5.84 + 4.90} \times 7.40\%$$
$$= 4.02\%$$

$$\text{Total read busy} = (\text{Read \% of seeking}) + \text{Read busy}$$
$$= 4.02\% + 8.60\%$$
$$= 12.62\%$$

$$\text{Write \% of seeking} = \frac{\text{Write rate}}{\text{Read rate + Write rate}} \times \text{Seek busy}$$
$$= \frac{4.90}{5.84 + 4.90} \times 7.40\%$$
$$= 3.38\%$$

$$\text{Total write busy} = (\text{Write \% of seeking}) + \text{Write busy}$$
$$= 3.38\% + 6.96\%$$
$$= 10.34\%$$

Disk demand for estimating response time is derived by subtracting the sum of the lowest total write busy from the total disk busy as follows:

$(38.10 + 45.23 + 34.83 + 41.10)$
$- (9.41 + 12.19 + 9.00 + 10.34)$
$= 159.26 - 40.94$
$= 118.32\ \%$

For *response*-time calculations, this results in a physical disk demand per logical I/O of:

$(118.32\ \% / 55.36) = 21.4$ ms

The physical disk demand for *utilization* calculations per logical I/O is:

$(159.26\ \% / 55.36) = 28.8$ ms

When estimating response time at the disk, both disk demands are needed; the first one for determining the demand in the numerator and the second for determining the utilization in the denominator.

Figure 4.
Seek apportioning for $D2.

Table 4.
Disk demands per transaction. (Misc. refers to MEASURE and simulator activity.)

Transaction type	Logical I/O per transaction	Disk demand per transaction for response time (ms)	Disk demand per transaction for utilization (ms)
TRAN1	6.19	132.47	178.27
TRAN2	1.00	21.40	28.80
TRAN3	3.04	65.06	87.55
Misc.	1.75 (total)	N/A	50.40 (total)

Figure 5

$$TRAN1\ R(disk) = \frac{D(disk)}{1 - U(average\ disk)} = \frac{132.47}{1 - 0.1991} = 165.40\ ms$$

$$TRAN2\ R(disk) = \frac{D(disk)}{1 - U(average\ disk)} = \frac{21.40}{1 - 0.1991} = 26.72\ ms$$

$$TRAN3\ R(disk) = \frac{D(disk)}{1 - U(average\ disk)} = \frac{65.06}{1 - 0.1991} = 81.23\ ms$$

Figure 6

$$R(TRAN1) = R(cpu) + R(disk) = 345.10 + 165.40 = 510.50\ ms$$

$$R(TRAN2) = R(cpu) + R(disk) = 147.53 + 26.72 = 174.25\ ms$$

$$R(TRAN3) = R(cpu) + R(disk) = 238.34 + 81.23 = 319.57\ ms$$

Figure 5.
Calculation of disk subsystem response time.

Figure 6.
Total response-time calculation using M/M/1.

For each transaction type, a physical disk demand for response time and a physical disk demand for utilization are calculated by multiplying the logical I/O rate per transaction by the specific demand per I/O from above. The results are shown in Table 4.

The total disk utilization for this system is 159.26 for eight spindles, an average of 19.91% (1.5926/8.00). All the information needed to obtain the disk subsystem's contribution to response time is now available. (See Figure 5 for the final calculation.)

The final step in determining the total response time is to add the contribution by the CPU subsystem to the contribution by the disk subsystem, as shown in Figure 6.

The actual response times are found in Table 1 (500 ms, 110 ms, and 300 ms for TRAN1, TRAN2, and TRAN3 respectively). The M/M/1 model comes fairly close in estimating the actual response times for TRAN1 and TRAN3. The error is 2.1% and 6.5%, respectively. The error for TRAN2 is 54.7%. Because the error in the model for TRAN2 is so high, obviously a straight M/M/1 model is not the correct model to use. If the error is incorporated into the model by using the relative change method, the results are improved.

The Relative Change Calculations

Use the following formula to determine the correction factor for each transaction type which will be applied to future estimates.

$$\text{correction factor} = \frac{\text{actual}}{\text{estimated}}\ \text{baseline response time}$$

For the transactions, TRAN1, TRAN2, and TRAN3, the correction factors are:

$$TRAN1 = \frac{500}{510.50} = 0.97943$$

$$TRAN2 = \frac{110}{174.25} = 0.63128$$

$$TRAN3 = \frac{300}{319.57} = 0.93876$$

Figure 7

Step 1—CPU utilization for each transaction type

The utilization law, $U = XD$ where X is transaction rate and D is demand per transaction is used. D is found in Table 1.

$U(tran1) = 2.79 \times 147.53 = 411.61$ ms/sec or 41.161%
$U(tran2) = 4.42 \times 63.07 = 278.77$ ms/sec or 27.877%
$U(tran3) = 15.34 \times 101.89 = 1562.99$ ms/sec or 156.299%

$U(simulator) = 22.55 \times 8.00 = 180.40$ ms/sec or 18.040%

To determine the interrupt work in the system, add up all the other activity and add an additional 19.6% to it.

$$U(interrupt) = 0.196 \times (41.161 + 27.877 + 156.299 + 18.040)$$
$$= 47.70\%$$

Adding up all the pieces results in the estimated total CPU utilization :

$U(CPU) = 41.161 + 27.877 + 156.299 + 18.040 + 47.70 = 291.077\%$

$Avg\ U\ (CPU) = 2.91077/4.00 = 0.7277$ or 72.77%

Step 2—Calculating total disk demand (or utilization)

Again the $U = XD$ formula is used. Since the intent is to find the average disk utilization, the parallel writes will be included. The D is found in the third column of Table 4.

$U(tran1) = 2.79 \times 178.27 = 497.37$ ms/sec or 49.73%
$U(tran2) = 4.42 \times 28.80 = 127.30$ ms/sec or 12.73%
$U(tran3) = 15.34 \times 87.55 = 1343.02$ ms/sec or 134.30%
$U(misc) = 50.40$ ms/sec or 5.04%

Adding up the pieces results in the estimated total disk utilization of:

$U(DISK) = 49.73 + 12.73 + 134.30 + 5.04 = 201.80\%$

$Avg\ U\ (DISK) = 2.0180 / 8.00 = 0.2523$ or 25.23%

Step 3—Calculating the M/M/1 response time

The formula for the M/M/1 estimated response time is:

$$\frac{D(cpu)}{1 - U(avg\ cpu)} + \frac{D(disk)}{1 - U(avg\ disk)}$$

Using the demands per transaction from the baseline and the new estimated average utilizations for both the CPU and the Disk, the response times are calculated. The calculations are as follows:

$$R(TRAN1) = \frac{147.53}{1 - 0.7277} + \frac{132.47}{1 - 0.2523} = 718.96 \text{ ms}$$

$$R(TRAN2) = \frac{63.07}{1 - 0.7277} + \frac{21.40}{1 - 0.2523} = 260.24 \text{ ms}$$

$$R(TRAN3) = \frac{101.89}{1 - 0.7277} + \frac{65.06}{1 - 0.2523} = 461.19 \text{ ms}$$

Step 4—Calculating relative change response-time estimate

Taking the M/M/1 response-time estimates and multiplying them by their respective "correction factors," the relative change estimate of response time is obtained. The calculations are as follows:

TRAN1 = 718.96 × 0.97943 = 704.17 ms
TRAN2 = 260.24 × 0.63128 = 164.28 ms
TRAN3 = 461.19 × 0.93876 = 432.95 ms

Results

The actual average CPU utilization was 73.30 vs. 72.77 estimated.

The actual average disk utilization was 27.48 vs. 25.23 estimated.

The actual response times as reported by the simulators at these transaction rates were:

$R(TRAN1) = 700$ ms
$R(TRAN2) = 140$ ms
$R(TRAN3) = 440$ ms

Forecasting with the Model

Figure 7 shows how to estimate response time at a different transaction rate using both the M/M/1 model and the relative change model. The new transaction rates are:

TRAN1 – 2.79
TRAN2 – 4.42
TRAN3 – 15.34

The steps involved in estimating the response time are as follows:

1. Determine total CPU utilization for each transaction type and other processes in the system. Determine total system CPU utilization and average utilization.
2. Determine total disk utilization for each transaction type and other processes in the system. Determine total system disk utilization and average utilization.
3. Calculate M/M/1 response-time estimate.
4. Calculate relative change response-time estimate.

Figure 7.
Using a response-time model for future transaction rates.

Table 5.
Comparison of estimates and their errors.

Transaction type	Actual response time	M/M/1 estimate	Relative change estimate	%error M/M/1	%error relative change
TRAN1	700	718.96	704.17	2.71	.60
TRAN2	140	260.24	164.28	85.89	17.35
TRAN3	440	461.19	432.95	4.82	- 1.60

Table 6.
Comparison of estimates and their errors at differing transaction rates.

Transaction type	Average CPU utilization	Actual response	M/M/1 estimate	Relative change estimate	%error M/M/1	%error relative change
TRAN1	80.69	0.870	0.960	0.891	10.34	2.41
TRAN2		0.160	0.340	0.215	112.50	34.37
TRAN3		0.580	0.592	0.556	2.07	- 4.14
TRAN1	45.78	0.420	0.432	0.423	2.86	0.71
TRAN2		0.100	0.143	0.090	43.00	-10.00
TRAN3		0.250	0.267	0.251	6.80	0.40
TRAN1	37.21	0.380	0.390	0.382	2.63	0.53
TRAN2		0.090	0.126	0.080	40.00	-11.11
TRAN3		0.220	0.239	0.224	8.64	1.82
TRAN1	31.72	0.360	0.367	0.359	1.94	- 0.28
TRAN2		0.090	0.117	0.074	30.00	-17.78
TRAN3		0.210	0.224	0.210	6.67	0.00

Note: The response times in the chart are expressed in seconds.

Table 5 contains the actual response time, the two different estimates and the percent errors, for all three transaction types. Note the improved accuracy of the relative change prediction over a straight M/M/1 model.

Response times were estimated and compared to the actual response time calculated for four additional transaction rates. A summary of the results, including the estimates of response time, the actual response time, the percent errors, and the actual average CPU utilization is presented in Table 6. Figure 8 shows a graphical representation of the results for each transaction type.

Conclusion

By using the relative change correction factor, the error was reduced from a high of over 100% to a high of 35%. This error (35%) was achieved at a high CPU utilization (80%). At high utilizations, the response-time estimates are asymptotic and heading toward infinity. At this point, the system cannot keep up with the amount of work being requested and the request queue keeps getting larger and larger. The system is no longer in a steady state. Without taking the predictions at this high utilization point, the estimates using the correction factor were within 18% of the actual response time.

The M/M/1 model presented in this article is simple. The improvement in accuracy by using the relative change correction factor is very encouraging. However, this technique does *not* apply to every Tandem system and it is critical to understand when and if this technique can be used. It should *only* be attempted by someone knowledgeable in computer modeling.

The validation of the model required that the response time be known. In some cases, this can be accomplished using MEASURE user-defined counters. In other cases, this metric cannot be measured.

One of the most important points of computer modeling is that validation of the model is a necessity. When equipment purchases are based on the output from an analytical model, it is extremely desirable that the model be accurate. If the model cannot be validated, it should not be used. Another approach, such as simulation, must be used.

References

Buzen, J., and Denning, P. 1978. The Operational Analysis of Queueing Network Models of Computer Systems. *ACM Computing Surveys*. Vol. 10, No. 3.

Allen, A.0. 1978. *Probability, Statistics and Queueing Theory with Computer Applications*. Academic Press.

Lazowska, E., et al. 1984. *Quantitative System Performance*. Prentice Hall.

Horwitz, H., Shugh, S., and Sitler, S. 1988. Performance Management and SURVEYOR (available through your local Tandem analyst). Part no. 15308. Tandem Computers Incorporated.

Kendall, D.G. 1985. Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of Imbedded Markov Chains. *Annals of Mathematical Statistics*. Vol. 24.

MEASURE Reference Manual. Part no. 82441. Tandem Computers Incorporated.

MEASURE User's Guide. Part. no. 82440. Tandem Computers Incorporated.

Acknowledgments

The author would like to thank Dr. Kenneth Sevcik, Karna Thulin, Allen Barr, and Scott Sitler for their guidance and assistance, and also the reviewers who provided comments and suggestions.

Helaine Horwitz joined Tandem in 1986 as a senior staff analyst. She is presently part of the Product Specialists Group in LSMS. Before coming to Tandem, Helaine worked as an application designer and performance specialist for an Alliance member. She received a B.S. in Management Science from Stevens Institute of Technology and an M.S. in Operations Research in 1984 from Polytechnic Institute in New York.

Figure 8

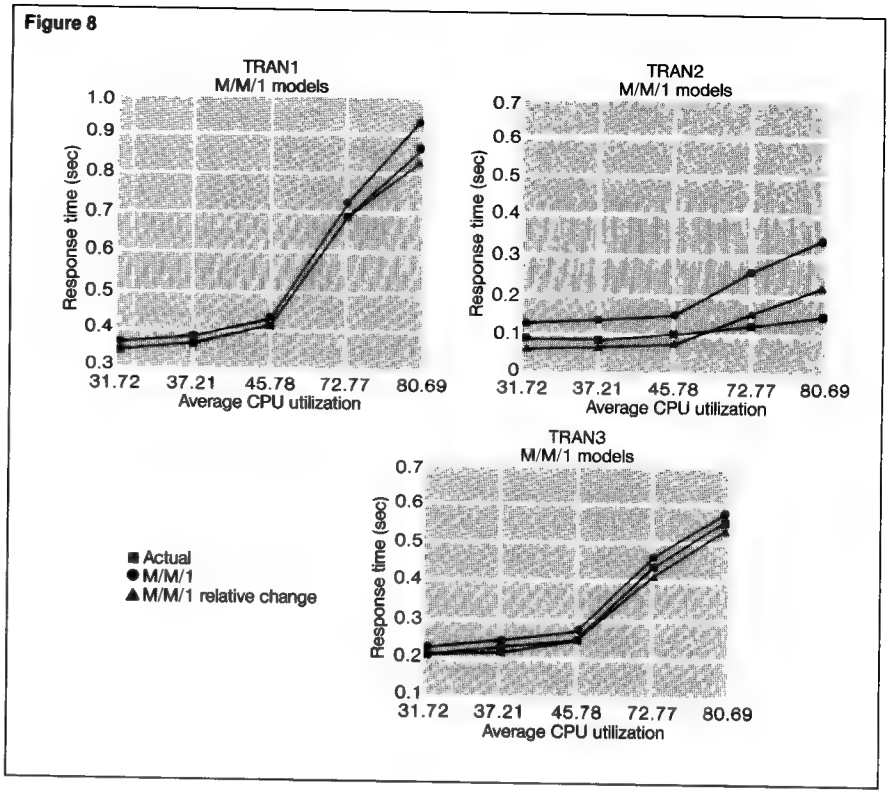


Figure 8.
Response time (estimated vs. actual).

TANDEM PUBLICATIONS ORDER FORM

Subscriptions to the *Tandem Systems Review* are free. Use this form to subscribe, change a subscription, and order back copies.

For requests *within the U.S.*, send this form to:

Tandem Computers Incorporated
Tandem Systems Review
18922 Forge Drive, LOC 216-05
Cupertino, CA 95014

For requests *outside the U.S.*, send this form to your local Tandem sales office.

Check the appropriate box(es):

- ☐ New subscription (# of copies desired _____)
- ☐ Subscription change (# of copies desired _____)
- ☐ Request for back copies. (Shipment subject to availability.)

Print your current address here:

COMPANY NAME

ADDRESS

ATTENTION

PHONE NUMBER (U.S.)

If your address has changed, print the old one here:

COMPANY NAME

ADDRESS

ATTENTION

PHONE NUMBER (U.S.)

To order back copies, write the number of copies next to the title(s) below. Please allow six to eight weeks for delivery.

NUMBER OF COPIES

Tandem Journal

_____ Part No. 83930, Vol. 1, No. 1, Fall 1983
_____ Part No. 83931, Vol. 2, No. 1, Winter 1984
_____ Part No. 83932, Vol. 2, No. 2, Spring 1984
_____ Part No. 83933, Vol. 2, No. 3, Summer 1984

Tandem Systems Review

_____ Part No. 83937, Vol. 2, No. 2, June 1986
_____ Part No. 83938, Vol. 2, No. 3, December 1986
_____ Part No. 83939, Vol. 3, No. 1, March 1987
_____ Part No. 83940, Vol. 3, No. 2, August 1987
_____ Part No. 11078, Vol. 4, No. 1, February 1988
_____ Part No. 13693, Vol. 4, No. 2, July 1988
_____ Part No. 15748, Vol. 4, No. 3, October 1988

TANDEM EMPLOYEES: PLEASE ORDER YOUR COPIES THROUGH YOUR MARKETING LITERATURE COORDINATOR.

10/88

TANDEM

Part No. 15748

400105 10/88 Printed in USA